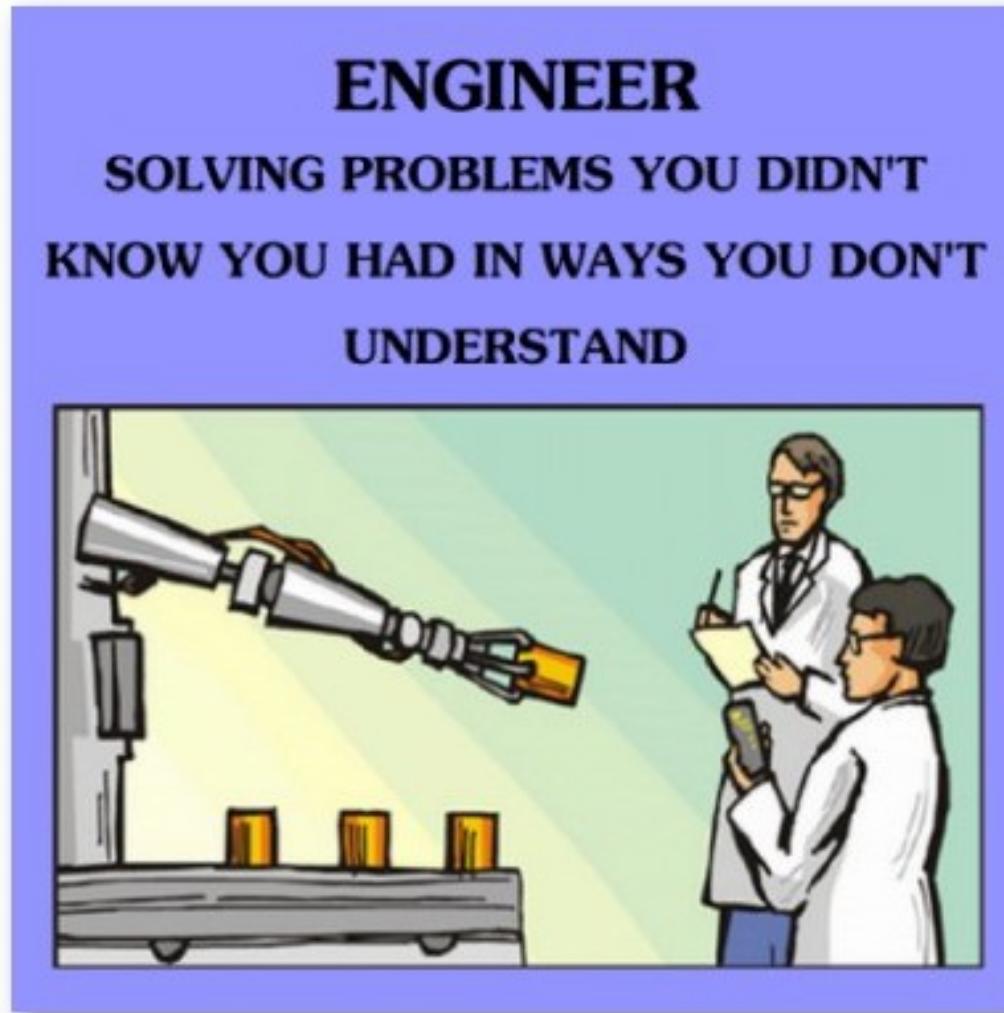


Software engineering

Introduction

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

What is Engineering?

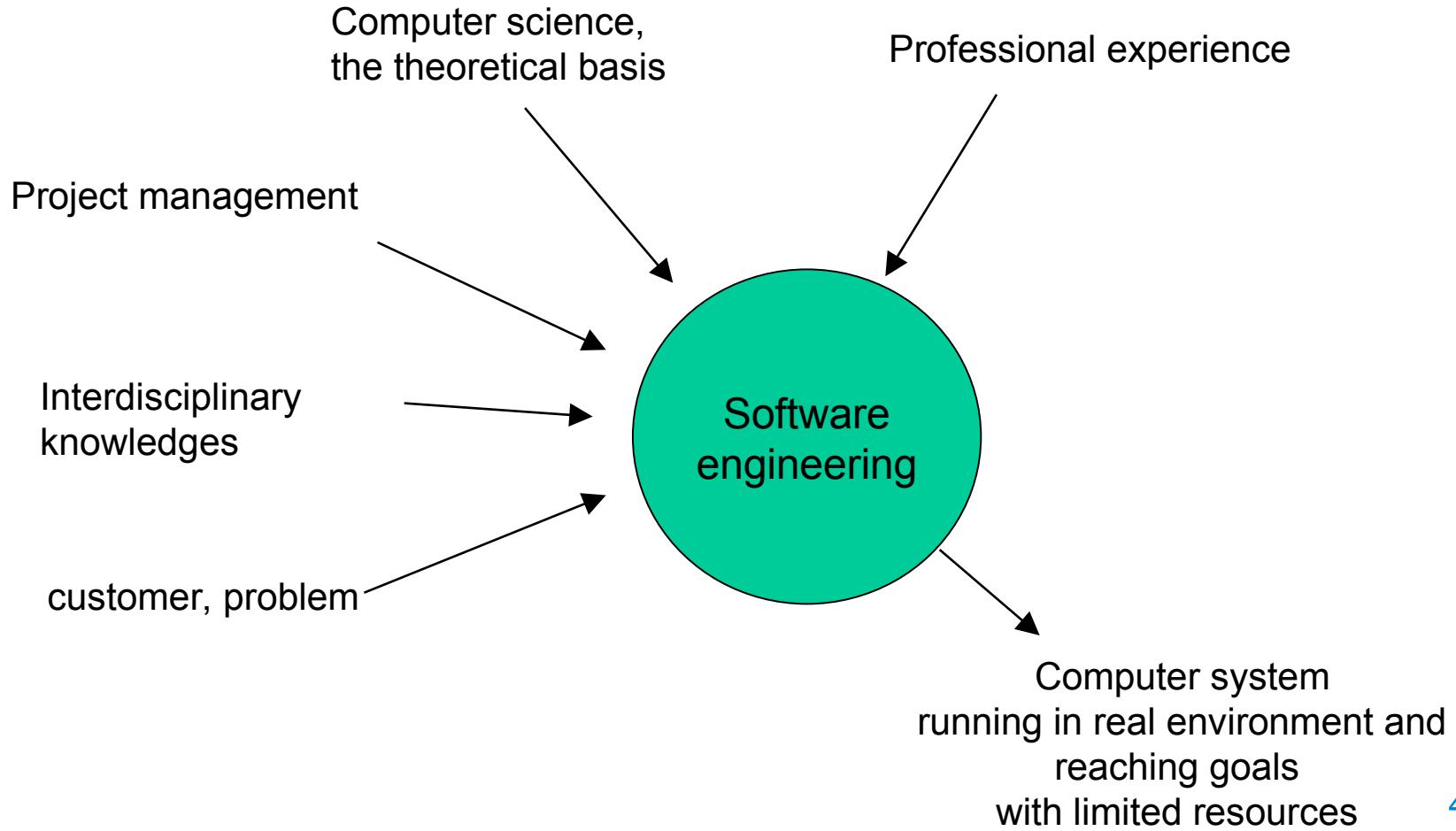


What is Software Engineering

- ❖ IEEE definition (1993):

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

SW engineering



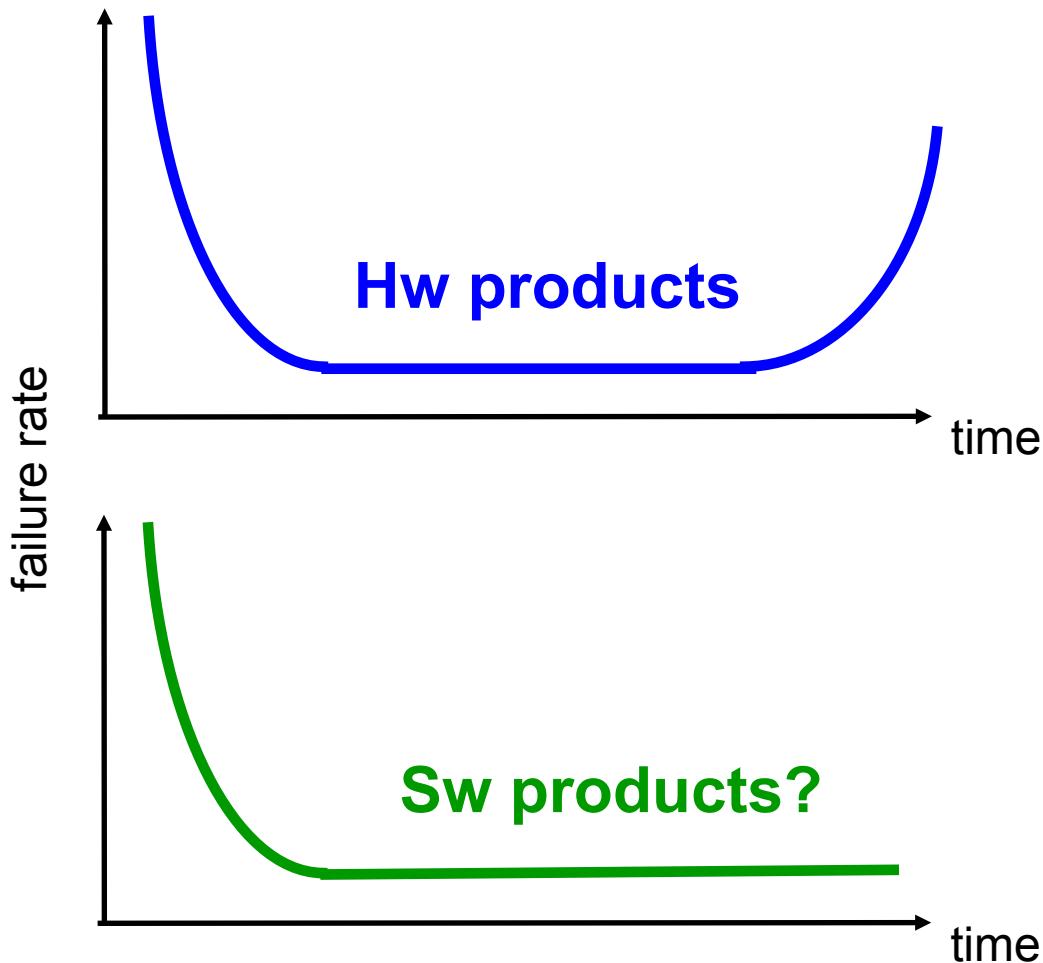
Software product

- ❖ **Computer programs** (software), which enables user to reach an effect wanted.
- ❖ **Data structures**, which enable SW to manipulate data (information).
- ❖ **Documents**, which describe the software from implementation and user perspective.

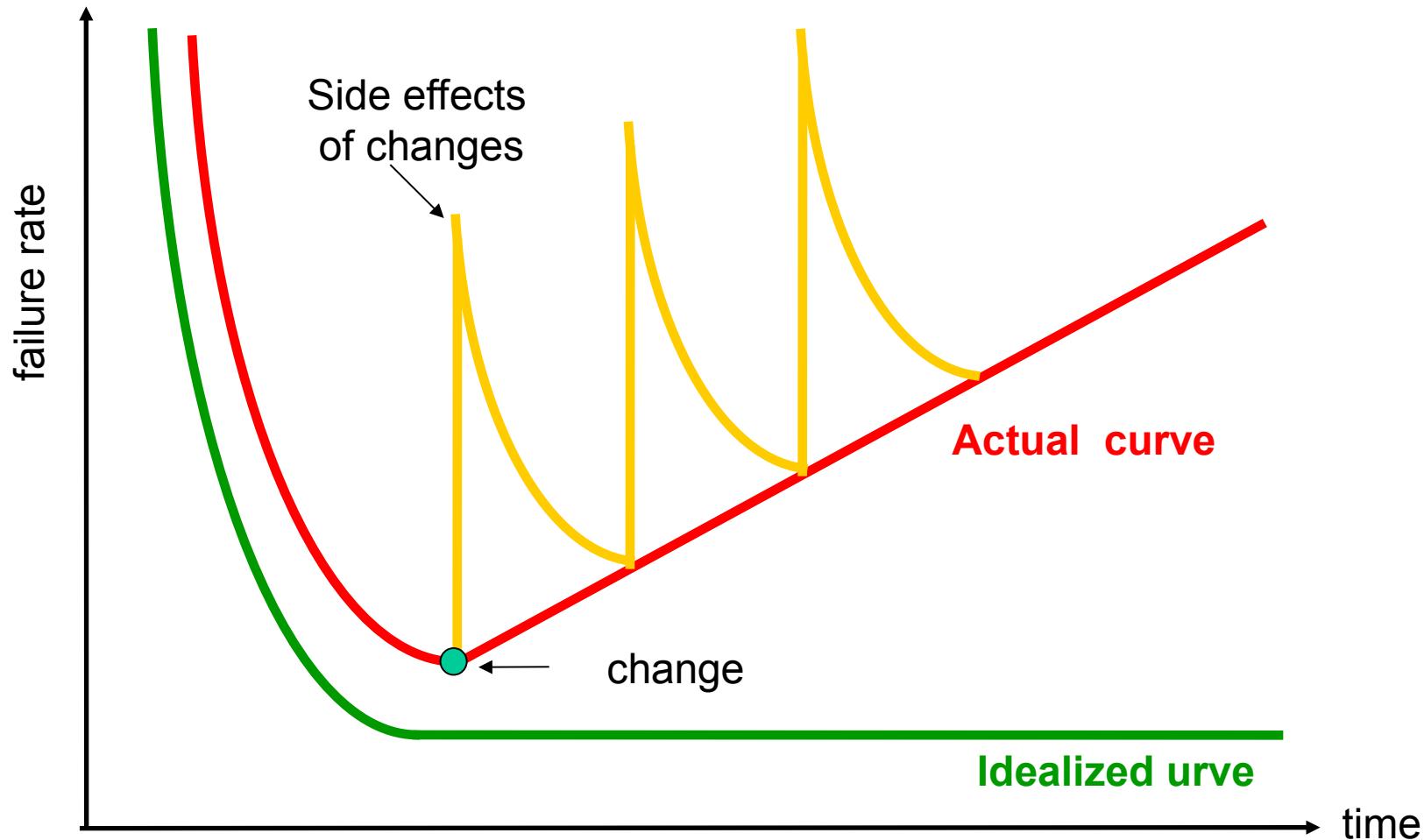
Software Engineering specialties

- ❖ SW includes hidden states: large number of invisible variables.
- ❖ SW is not “produced” in the classic sense – it is “developed”.
- ❖ Most of the SW is custom built.
- ❖ SW is often the most expensive part of a whole system.
- ❖ Difficult measuring od development progress.
- ❖ Maintenance means error correction, improvements, additional functionalities.
- ❖ SW does not wear out. But its relative quality does decrease during maintenance and changes.

Failure rate



Failure rate



SW products and failures

- ❖ Software products do include defects.
- ❖ Software engineering offers techniques to reduce the number and fatality of defects/bugs.
- ❖ But we can (never?) avoid them...
 - ◆ *“...building software will always be hard. There is inherently no silver bullet.” F. Brooks*

Examples of terminal faults

❖ Vasa warship

- ◆ Ordered by Swedish king Gustav.
- ◆ Requirements were never clearly stated and had been changing during the construction.
- ◆ Ship sunk on her maiden voyage. It heeled over under the force of wind.



Examples of terminal faults

❖ Ariane 5

- ◆ Self destructed 37 seconds after take off (including 4 expensive uninsured research satellites) because of a malfunction in the control software.
- ◆ Error in the code that was not needed for Ariane 5 and was written for the Ariane 4.
- ◆ Software exception at data conversion from 64-bit floating point value to 16-bit signed integer value to be stored in a variable representing horizontal bias caused a processor trap (operand error) because the floating point value was too large to be represented by a 16-bit signed integer.

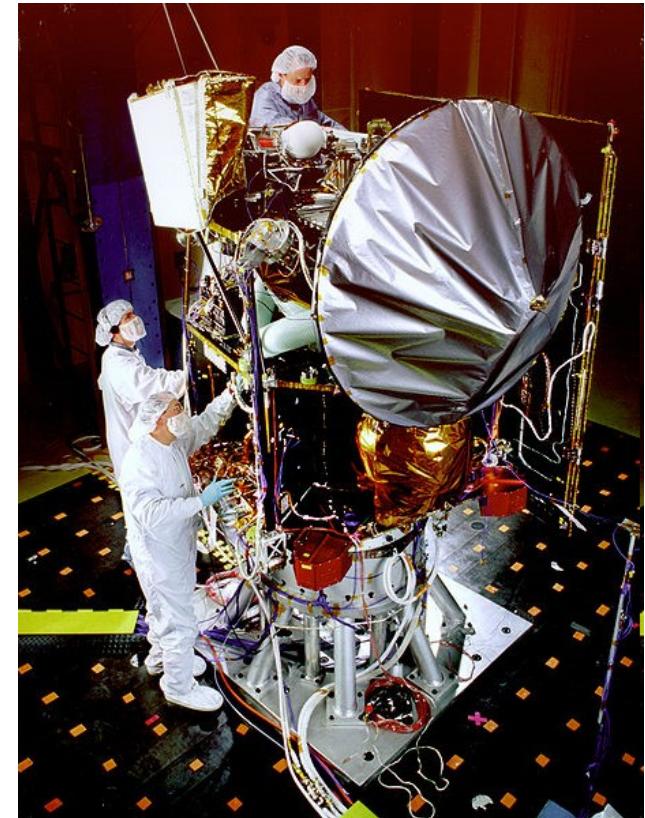
<http://www.around.com/ariane.html>



Examples of terminal faults

❖ Mars Climate Orbiter (1999)

- ◆ communication with the spacecraft was lost as the spacecraft went into orbital insertion, due to ground-based computer software which produced output in non-SI units of pound-force seconds ($\text{lbf}\cdot\text{s}$) instead of the SI units of newton-seconds ($\text{N}\cdot\text{s}$) specified in the contract between NASA and Lockheed. The spacecraft encountered Mars on a trajectory that brought it too close to the planet, and it was either destroyed in the atmosphere or re-entered heliocentric space after leaving Mars' atmosphere.



Vir: http://en.wikipedia.org/wiki/Mars_Climate_Orbiter

Examples of serious faults

❖ Airbus A330,

- ◆ Qantas Flight 72, 7.10.2008
- ◆ Sudden uncommanded pitch-down manoeuvres that severely injured many of the passengers and crew.
- ◆ Successful emergency landing
- ◆ After detailed forensic analysis of the FDR data, the flight control primary computer (FCPC) software and the air data inertial reference unit (ADIRU), it was determined that the CPU of the ADIRU corrupted the angle of attack (AOA) data. The exact nature was that the ADIRU CPU erroneously relabelled the altitude data word so that the binary data that represented 37,012 (the altitude at the time of the incident) would represent an angle of attack of 50.625 degrees. The FCPC then processed the erroneously high AOA data, triggering the high-AOA protection mode, which sent a command to the electrical flight control system (EFCS) to pitch the nose down. (fault of a velocity sensor)
- ◆ Airbus issued a SW correction that prevents such scenarios.



Examples of serious faults

Catastrophic software errors doomed Boeing's airplanes and nearly destroyed its NASA spaceship. Experts blame the leadership's 'lack of engineering culture.'

Morgan McFall-Johnsen Feb 29, 2020, 2:07 PM



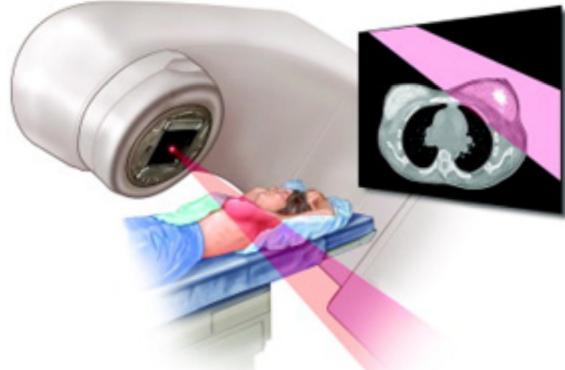
Left: Right: The CST-100 Starliner spacecraft is prepared for Boeing's Orbital Flight Test, November 2, 2019. Jason Redmond/Reuters; Boeing

<https://www.businessinsider.com/boeing-software-errors-jeopardized-starliner-spaceship-737-max-planes-2020-2>

Examples of serious faults



One of the last photos transmitted by Beresheet, the first private mission to attempt a moon landing. The spacecraft was designed and built by the Israeli nonprofit Spacell. Spacell/IAI via YouTube



OCTOBER 3, 2018

Security failure at Facebook—what we know



The Facebook breach affecting some 50 million users and



Why did Google Plus Fail as a Social Network?



Success of software products

- ❖ Success rate of software products is lower than success rate in other engineering disciplines!
- ❖ SW product can be successful if
 - ◆ The development is completed
 - ◆ Is useful
 - ◆ Is usable
 - ◆ Is used

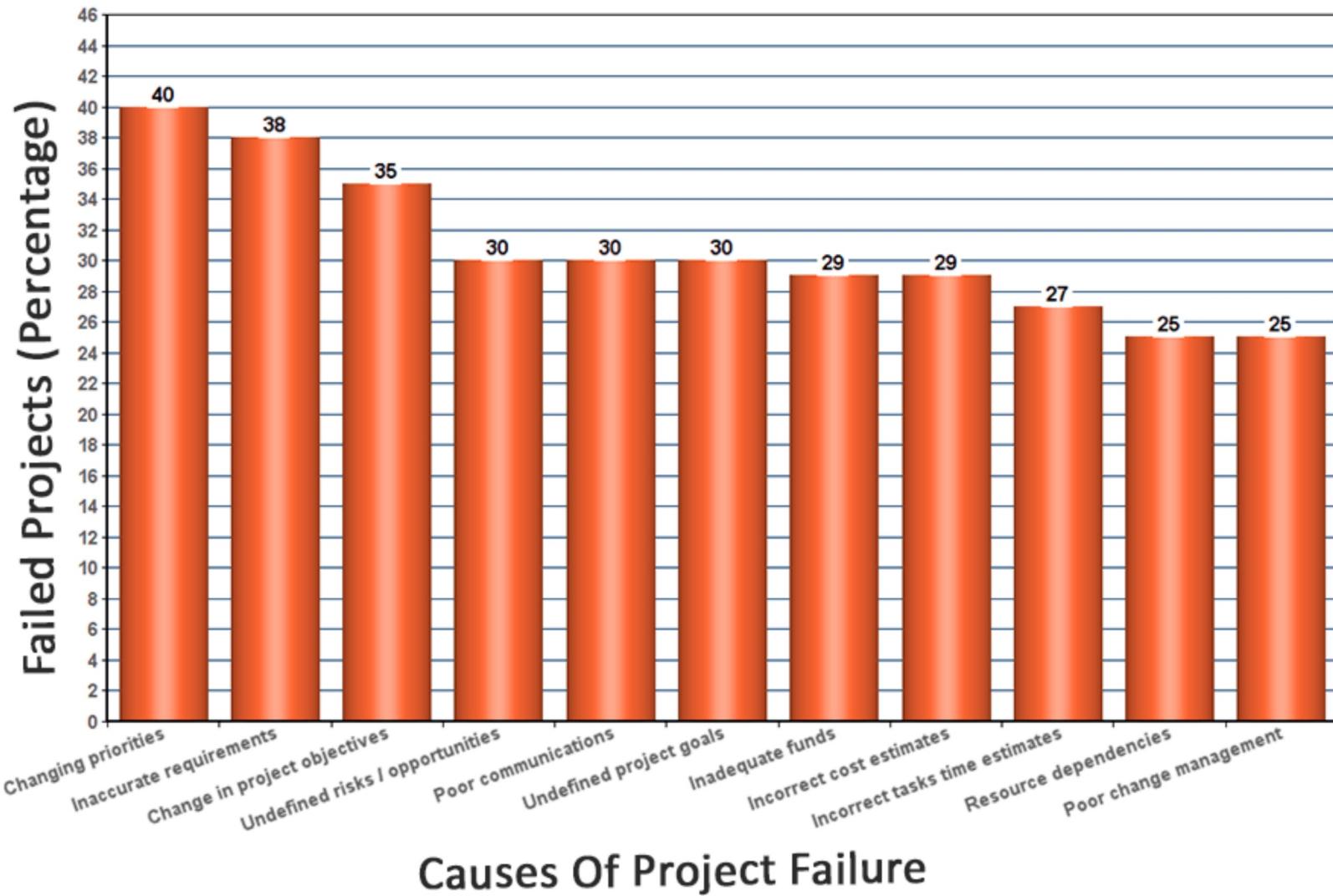
Reasons for failure

- ❖ Quality issues – the system is faulty (bugs)
- ❖ Development does not meet deadlines
- ❖ Over budget (too expensive)
- ❖ Does not fulfill user needs.
- ❖ Difficult to maintain.

Reasons for failures

- ❖ Inadequate development planning (e.g. no milestones defined)
- ❖ Undefined project goals
- ❖ Poor understanding of user requirements
- ❖ Insufficient quality control (testing)
- ❖ Technical incompetence of developers
- ❖ Poor understanding of cost and effort by both developer and user

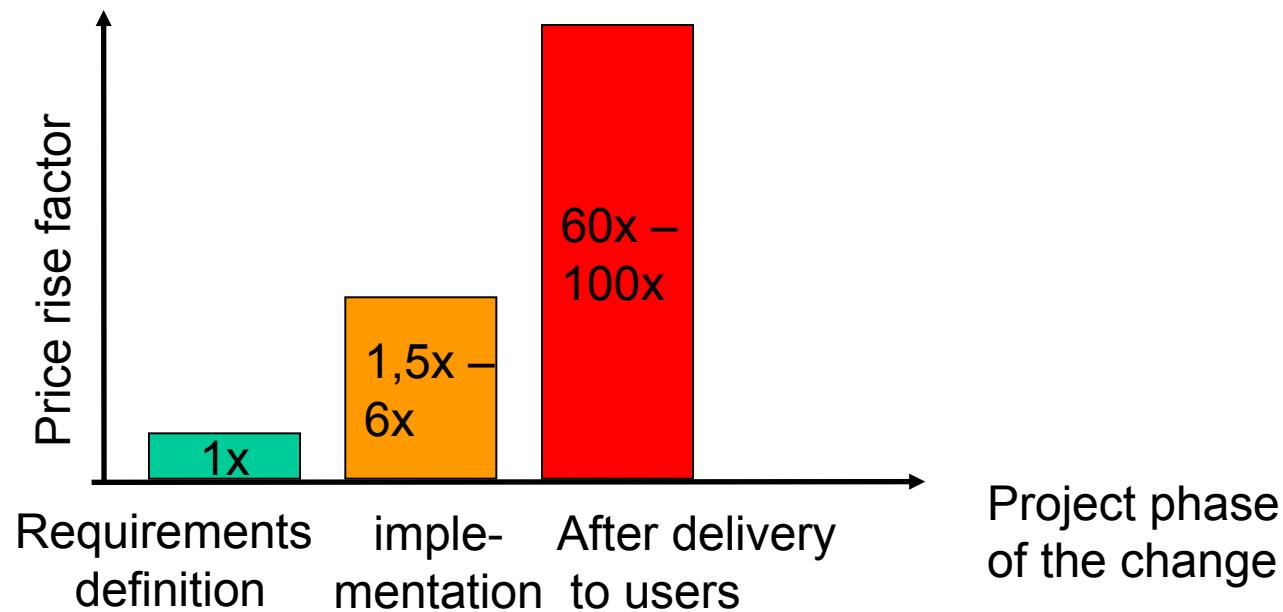
Why software projects fail



Source: <https://www.thesunflowerlab.com/blog/8-challenges-affecting-custom-software-development-project-management/>

SW product cost

- ❖ The major part of the cost are personnel costs (developer work)
- ❖ Changes in the project cause rise of costs.



Requirements to succeed

- ❖ Project Sponsorship at executive level
- ❖ Strong project management
- ❖ The right mix of team players
- ❖ Good decision making structure
- ❖ Good communication
- ❖ Team members are working toward common goals

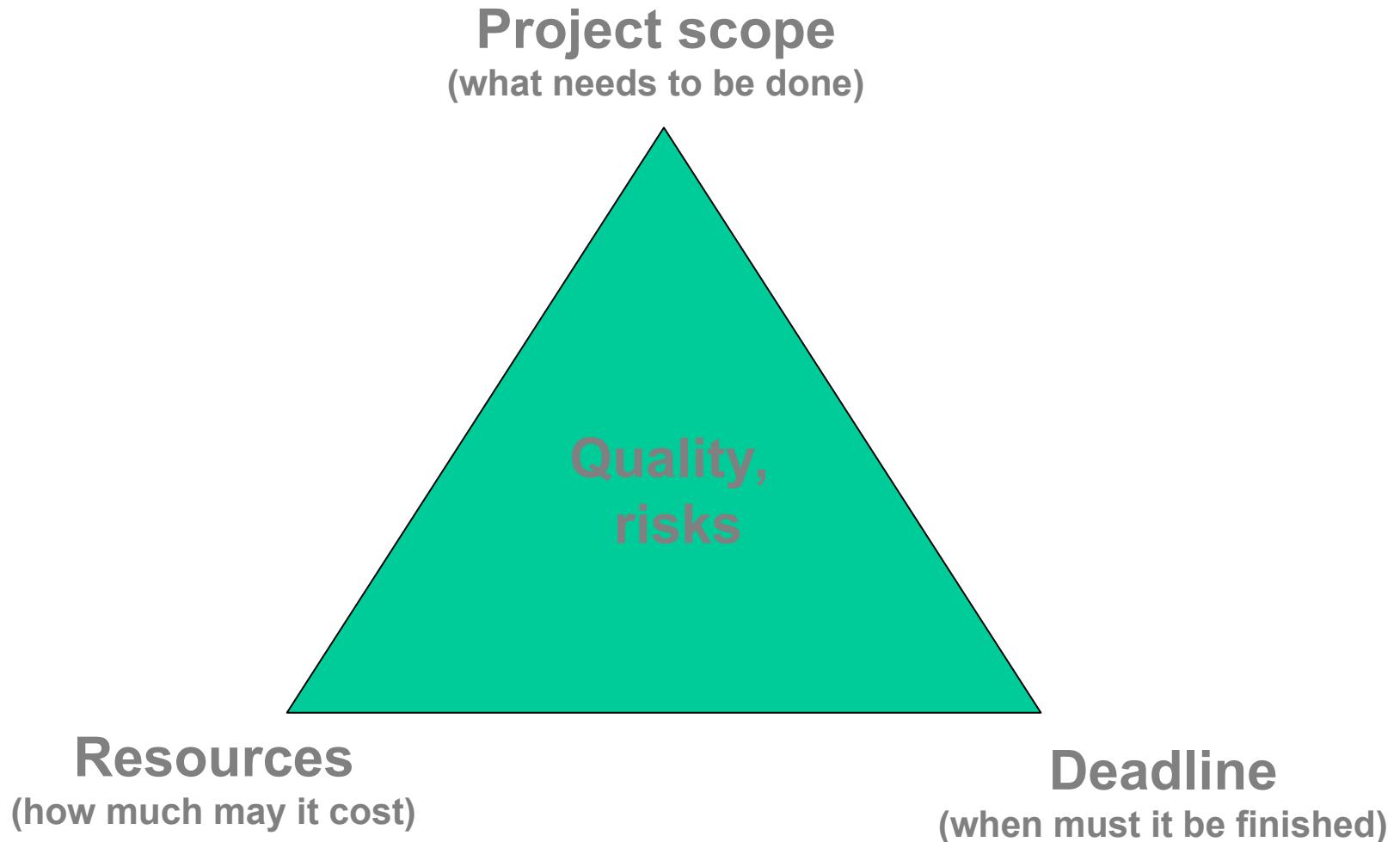
Characteristics of good SW products

- ❖ **Functionality** – to fulfill user needs
- ❖ **Maintainability** – ability to further develop SW due to changed requirements.
- ❖ **Reliability** – SW worth to trust.
- ❖ **Efficiency** – usage of system resources.
- ❖ **Acceptability** – ability to be accepted by users for who it was developed: understandable, useful, compatible with other systems.

Engineering principles

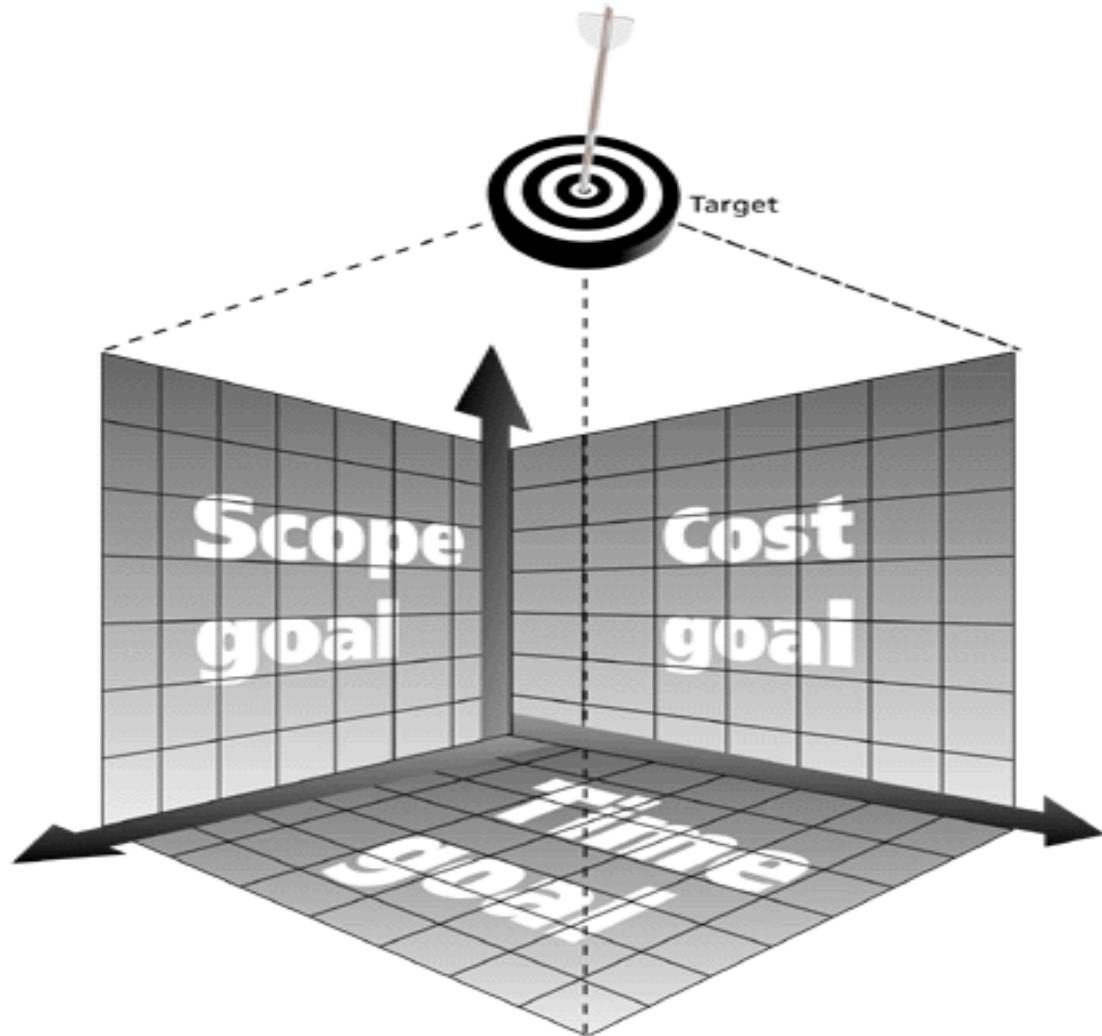
- ❖ Evaluation of costs
- ❖ Time planning
- ❖ Inclusion of users when defining requirements
- ❖ Defining development phases
- ❖ Defining of end products
- ❖ Following and managing the project progress
- ❖ Designing of quality control
- ❖ Exhaustive testing

Project limitations and goals



Limitations and project goals

A successful project must respect all three limitations. This is responsibility of a project manager.



Big/complex project challenges

- ❖ Development requires a lot of effort
- ❖ High development cost
- ❖ Longer project duration
- ❖ Changing user needs
- ❖ High risk of failure: poor user acceptance, performance, difficult maintenance...

Size does matter



Software project actors

- ❖ Project sponsor
- ❖ Project manager
- ❖ Project team
- ❖ Supporting personnel
- ❖ Customers
- ❖ Users
- ❖ Suppliers
- ❖ Opponents



How the customer explained it



How the Project Leader understood it



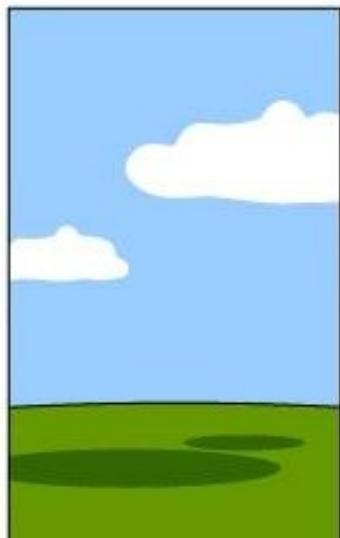
How the Analyst designed it



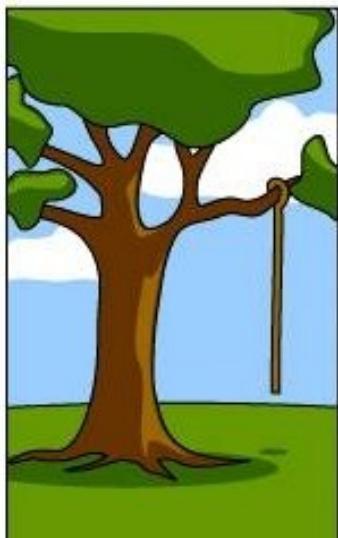
How the Programmer wrote it



How the Business Consultant described it



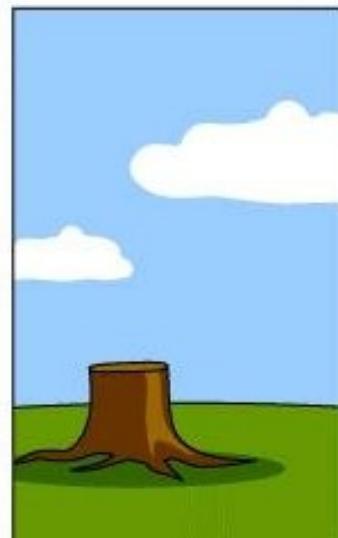
How the project was documented



What operations installed



How the customer was billed



How it was supported



What the customer really needed

SW project manager challenges

❖ Considering users

- ◆ Poorly defined requirements
- ◆ Must be focused to simplicity of use and responsiveness.

❖ Considering the development team

- ◆ Focused to development principles, technical solution.

❖ Considering executive management

- ◆ Focused to economic success. Costs and time limitations.

SW engineer challenges

- ❖ Heterogeneity
 - Development techniques for different platforms and environments
- ❖ Development speed
 - ◆ Techniques that enable fast sw development and early delivery to users.
- ❖ Trust
 - ◆ Techniques worthy to trust by the users.

Etics

- ❖ SW engineering involves wider responsibilities than only application of technical skills:
 - **Confidentiality.** Spoštovanje zaupnosti in varovanje podatkov delodajalcev in strank.
 - **Competence.** Engineers should not misrepresent their level of competence. They should not knowingly accept work which is outwith their competence.
 - **Intellectual property rights.** Engineers should be aware of local laws governing the use of intellectual property such as patents, copyright, etc. They should be careful to ensure that the intellectual property of employers and clients is protected.
 - **Computer misuse.** Software engineers should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).

Ethical dilemmas

- ❖ Disagreement in principle with the policies of senior management.
- ❖ Your employer acts in an unethical way and releases a safety-critical system without finishing the testing of the system.
- ❖ Participation in the development of military weapons systems or nuclear systems
- ❖ ...

Software myths

- ❖ SW development requires modern hardware.
- ❖ Being behind the schedule can be compensated with additional resources – more programmers (Mongolian horde concept)
- ❖ An expressed wish of a customer is sufficient to start coding – details can be considered later.
- ❖ Project requirements change frequently, fortunately it is easy to cope with this as software development offers plenty of flexibility.
- ❖ Our job is finished when coding is completed and software starts working.
- ❖ As long as the software cannot be executed not run, we cannot assess its quality.
- ❖ The only product of a software project is an executable software.
- ❖ SW engineering requires excessive documenting, which slows down the development.

Software engineering

Software process

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Software process

- ❖ Software process is a process of building a software product from detected need till the end of maintenance.
- ❖ In software process we face with technical and organizational challenges.
- ❖ The result of a software process is a software product.
- ❖ Different software products require different software processes.

Basic presumptions

- ❖ A good software process leads to a good software product.
- ❖ Good software process reduces risks.
- ❖ Good software product is more manageable and clear.

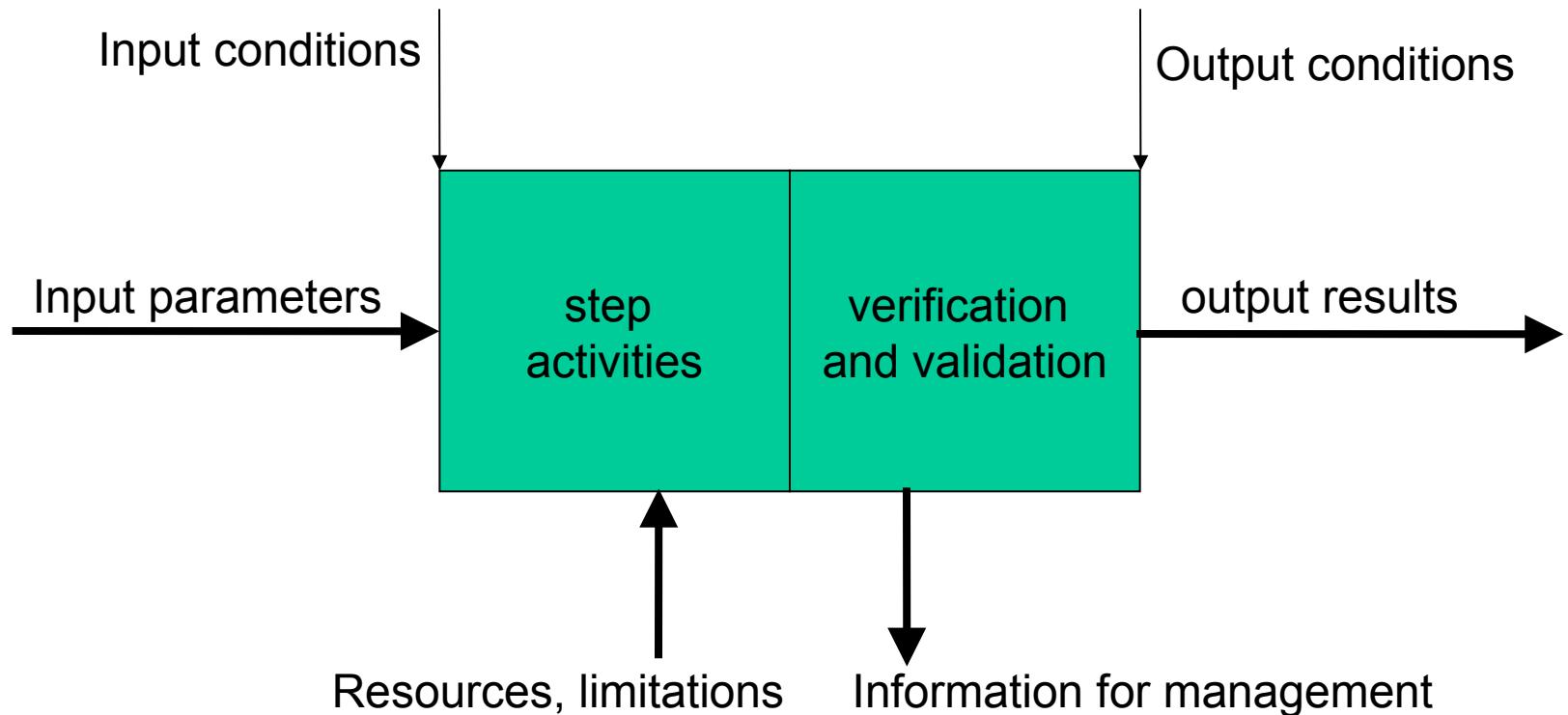
A structure of a software process

- ❖ A software process consists of activities/steps that need to be performed in the correct order.
- ❖ Each individual step
 - ◆ Must have clearly defined goals
 - ◆ Requires people with specific skills
 - ◆ Is based on clear and predefined input parameters
 - ◆ Results in clearly defined output results
 - ◆ Has a defined beginning time (and requirements to begin) and end time (and requirements to end).
 - ◆ Uses specific techniques, tools, guidelines agreements, standards, etc.

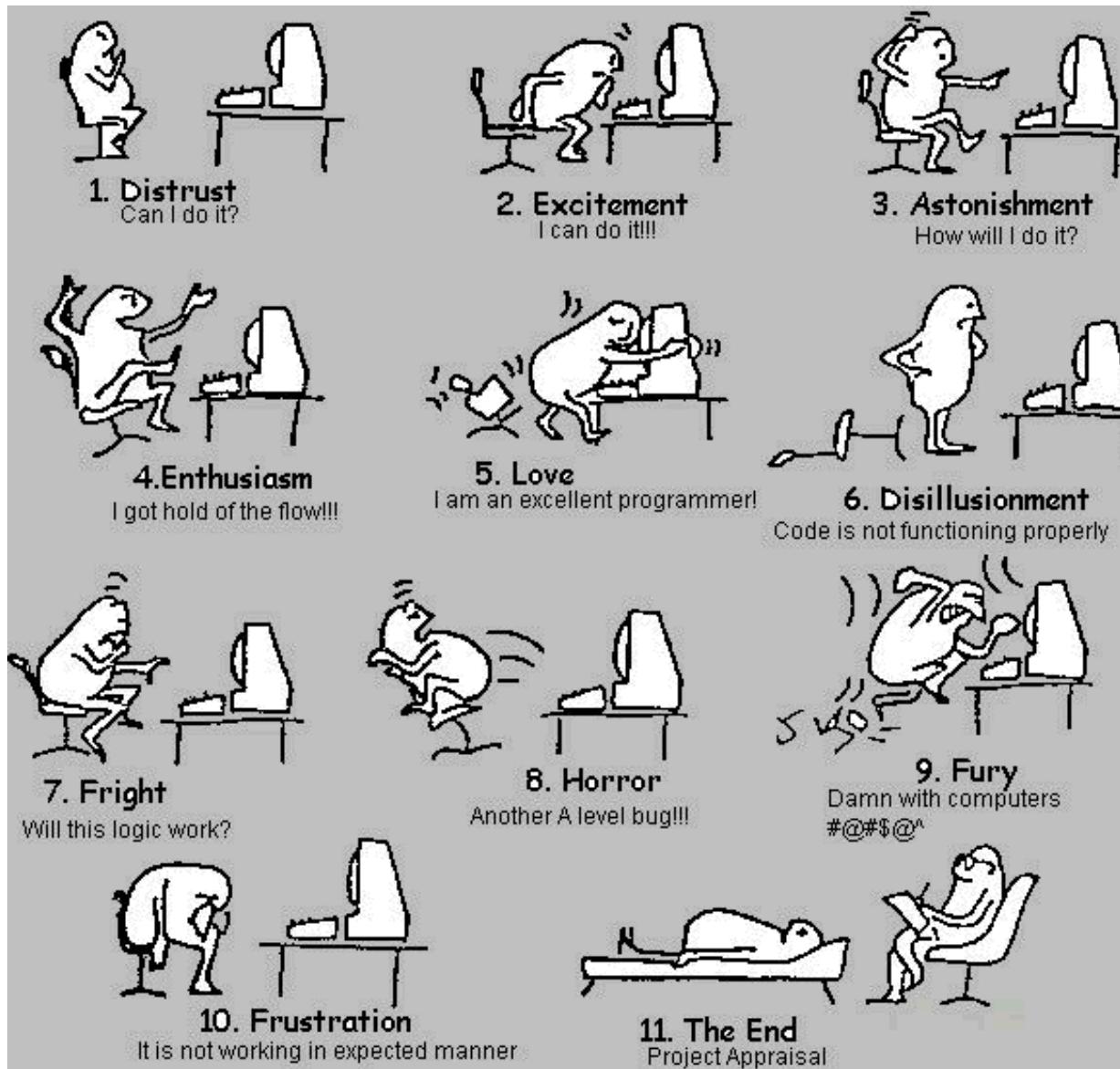
A software process step

- ❖ Must follow the project plan, which defines duration, resources, limitations, etc.
- ❖ It must provide information for management enabling timely corrective measures.
- ❖ Each step ends with its verification and validation part (V&V):
 - ◆ Verification – check consistency of outputs with input parameters.
 - ◆ Validation – check consistency with user needs.

A software process step



Software process phases ☺



Software process phases

- ❖ Problem definition
- ❖ Feasibility study
- ❖ Requirements analysis
- ❖ System modeling
- ❖ Component modeling
- ❖ Implementation
- ❖ Testing
- ❖ Deployment*
- ❖ Maintenance

* Could also be just a milestone between testing and maintenance

Problem definition 1/4

❖ Questions answered:

- ◆ **What is a problem?**
- ◆ Who experiences a problem and where?
- ◆ Why is there a problem at all?
- ◆ How can a problem be further explored?
- ◆ Users and management must agree about the problem.
- ◆ Avoid ambiguous definitions (multiple possible interpretations).

❖ This is a short step and requires a day or two.

Problem definition 2/4

- ❖ If the problem exists, if it needs to be solved and if resources are available to solve it, then it can later become a project.
- ❖ An estimation of costs of the next step - feasibility study – is needed.
 - ◆ Also, a rough estimation of project costs is needed, such that a user/customer gets a feeling of a project size. A better cost estimate will be made later.

Problem definition 3/4

- ❖ Different problem types are possible, e.g.:
 - ◆ Insufficient performance of an existent system,
 - ◆ The existent solution is not economical due to operational costs
 - ◆ Insufficient accuracy of existent solution
 - ◆ Insufficient reliability of existent solution
 - ◆ Insufficient safety of existent solution
 - ◆ Insufficient security of existent solution
 - ◆ Existing solutions do not enable obtaining of key information
 - ◆ The need for new functionality.
- ❖ What is a problem and what a solution!?
 - ◆ “A computer system for...” is a solution and not problem!

Problem definition 4/4

- ❖ The result of problem definition phase is a document - called problem definition which is typically 1-2 pages long:
 - ◆ Project name,
 - ◆ Concise problem description,
 - ◆ Project goals,
 - ◆ Eventual initial implementation ideas (optional),
 - ◆ Time needed for the next phase – feasibility study.
 - ◆ Rough cost estimates of the overall project costs.

Feasibility study 1/5

- ❖ Answers the questions:
 - ◆ **Is the project technically feasible?**
 - ◆ **Is the project economically feasible?**
 - ◆ **Is the project operationally feasible?**
 - ◆ Which are advantages provided by the project?
 - ◆ What is the extent of the project?
- ❖ There may be multiple alternatives proposed and documented.
- ❖ Feasibility study enables decision to start the project or not.

Feasibility study 2/5

- ❖ **Technical feasibility:**
 - ◆ For each alternative implementation technical details have to be considered for all the following phases (including maintenance) and estimate the risks.
 - ◆ Technical personnel must be included (those that may later become a part of a project team).

Feasibility study 3/5

❖ Economic feasibility

◆ Costs:

- One time costs of equipment (HW+SW), education, SW, consultations, support...
- Variable costs, salaries, assets, maintenance, rents...

◆ Benefits:

- Savings (salary savings, material, increase of production, reduced maintenance and usage costs...)
- Contributions (better customer services, improved management capabilities, control of production, reduction of failures and errors...)

Feasibility study 4/5

❖ How to estimate project costs?

- ◆ By splitting the system into components. Easier to do estimation for smaller components.
- ◆ Using historical data (previous projects)
- ◆ Estimation of personnel costs (development and maintenance). A time plan of cost changes shall be made.
- ◆ Using organizational standards to estimate additional costs (management, administration, rooms, electricity...)

Feasibility study 5/5

- ❖ The result of the feasibility study is a feasibility report, which is given to the management and to users. They have to estimate:
 - ◆ Are alternatives acceptable?
 - ◆ Are solving the right problem?
 - ◆ Whether any of alternatives offers advantages over the others or assures project success? Consideration of the expected SW lifespan is needed, typically 5-7 years.
- ❖ Users and management select one of the alternatives if they decide to start the project.
- ❖ Many projects never start and 'die' earlier -at this point.

Requirements analysis 1/6

- ❖ The aim of requirements analysis and definition is to describe the system from user perspective in details:
 - ◆ Functional requirements: use cases for all interactions of user with the system.
 - ◆ Other, nonfunctional, requirements, that limit design or implementation:
 - Operational requirements (security, safety, usefulness, performance...)
 - Evolution requirements (testing ability, maintenance ability, extensibility, upgradability...)
- ❖ Requirements analysis is performed by system analyst.
- ❖ Incorrect, incomplete or ambiguous specification often leads to project failure or disputes.

Requirements analysis 2/6

❖ Challenges:

- ◆ Users often do not know well or cannot describe their needs, or they do not know how SW could help.
- ◆ User wishes change with time.
- ◆ Users have inconsistent requirements.
- ◆ Users have difficulties estimating what is feasible (also related to costs)
- ◆ Users do not distinguish between wishes and requirements
- ◆ System analyst has only a limited knowledge of the system's field of use.
- ◆ Customer and user may not be the same person.

Requirements analysis 3/6

- ❖ The process:
 - ◆ Obtain knowledge from the field of use.
 - ◆ Getting acquainted with the problem (interviews, questionnaires, searching of inter-dependencies, identifying gaps).
 - ◆ Study of existing systems.
 - ◆ The analysis often from outside in – from required outputs to required inputs, processes, storage needs.
 - ◆ System description (describing a system textually, with diagrams...)
 - ◆ Recension of the requirements analysis, together with users.
 - ◆ This process may continue with another iteration of all mentioned here.

Requirements analysis 4/6

- ❖ The result is a document called Software requirements specifications (SRS), (sometimes incorrectly called functional specifications -FS)
 - ◆ Informs users of the system such that they can estimate if the proposed system is correctly addressing their requirements, if the developer understanding of a problem is correct.
 - ◆ Splits the problem into smaller peaces – components.
 - ◆ The document is later used for system design.
 - ◆ The document is later used for system validation.

Requirements analysis 5/6

- ❖ Document contents must be adjusted to fit the project needs and typically include:
 - ◆ Introduction (purpose, extent, definitions, references to other documents)
 - ◆ General description (relations to other systems, function and component overview, user specifics, general limitations)
 - ◆ Description of requirements for each of the components (description, inputs, processing, outputs)
 - ◆ External interface requirements (formats, HW, connections/communication to other systems)
 - ◆ Required performance
 - ◆ Design limitations (standards, HW limitations...)
 - ◆ Other requirements

Requirements analysis 6/6

- ❖ The requirements analysis phase ends with review:
 - ◆ Review of all requirements:
 - Validity. Do system functions comply with user needs?
 - Consistency. Are there requirements that oppose each other?
 - Completeness. Are all the functions needed by users considered?
 - Reality. Can requirements be satisfied with the proposed technology and with the cost limitations?
 - ◆ Review of individual requirements:
 - Testability. Can the fulfillment of the requirement be checked?
 - Understandability. Is the requirement correctly understood?
 - Traceability. Is it clear why certain requirement exist?
 - Adaptability. Can the requirement be changed or modified without high influence on other requirements?

System design 1/2

- ❖ In the phase of system design a **system architecture** is defined:
 - ◆ Structure of software components,
 - ◆ External properties of each component
 - ◆ Component relationships.
- ❖ Division of a whole system requirements into requirements of individual components (definition of functions).

System design 2/2

- ❖ Division of a system to components (decomposition)
 - ◆ Divide the system such that components are as much as possible independent – important for the maintenance.
 - ◆ Decomposition is vertical and horizontal:
 - **Partitions** vertical division into multiple weakly connected subsystems, which offer services at the same level of abstraction.
 - **Layers** are horizontal division to subsystems where higher layers use services of lower layers and offer services to higher layers.
 - ◆ System components include processing and data components

Component design 1/2

- ❖ Based on selected system design each individual component is further designed.
 - ◆ Component structure
 - ◆ Modules that need to be implemented
 - ◆ Database and file formats.
- ❖ Each component is defined with such level of details that it enables component implementation.

Component design 2/2

- ❖ Products of the component design are:
 - ◆ Component specifications (diagrams, pseudo-code, descriptions)
 - ◆ Data file design (organization, access principles...)
 - ◆ HW specification
 - ◆ Testing plan
 - ◆ Time plan for implementation
- ❖ The phase ends with a technical review.

Implementation 1/2

- ❖ In the implementation phase the software products is built:
 - ◆ A collection of components (SW) that can be:
 - Developed for that specific purpose
 - Obtained from elsewhere
 - Modified existing components
 - ◆ Data structures that enable SW to manipulate with (process) data.
 - ◆ Documentation that describe operation and usage of SW.

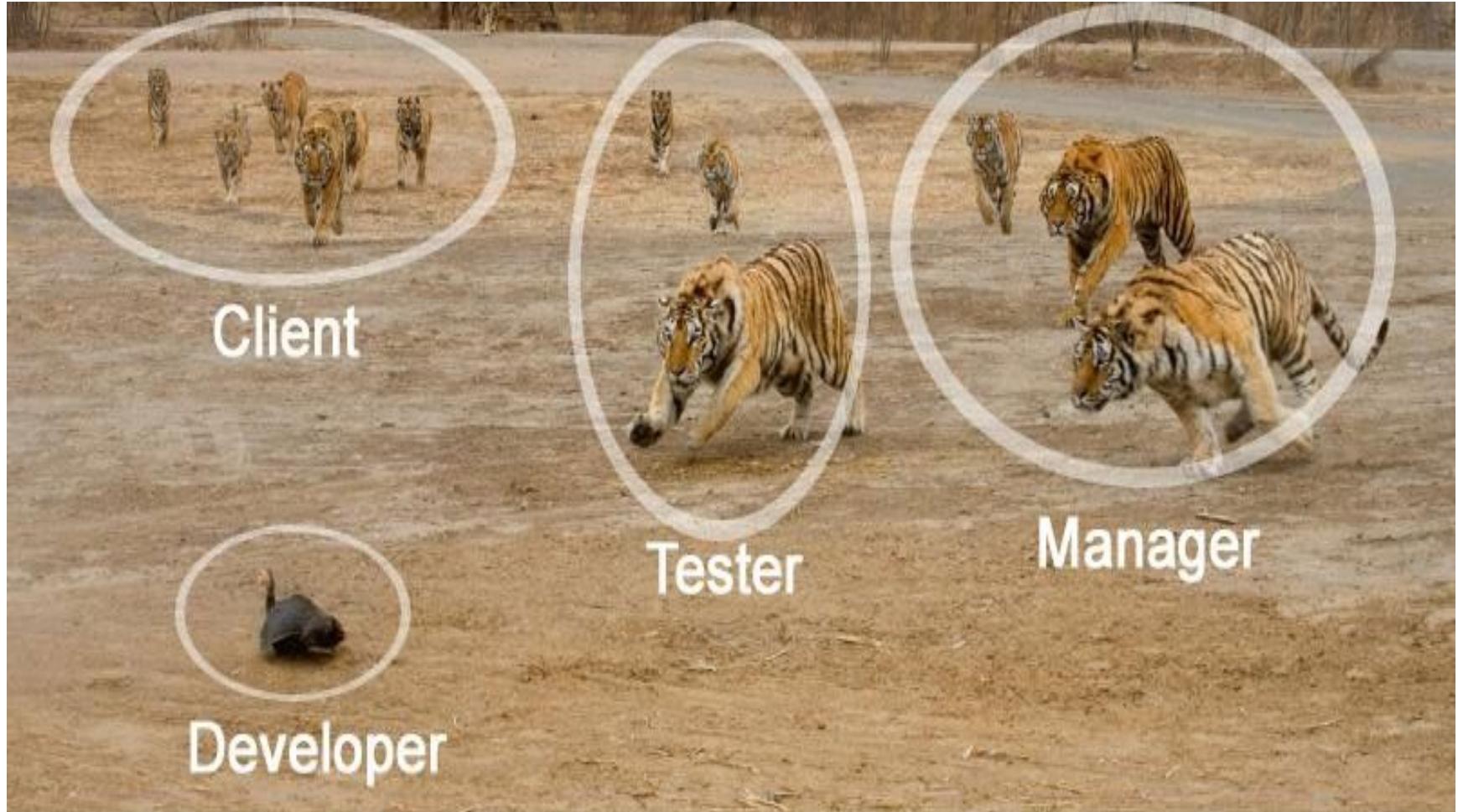
Implementation 2/2

- ❖ Implementation ends with verification:
 - ◆ Individual components are tested if they comply with component specifications.
 - ◆ Components are integrated into a software system and tested against the system specifications.

Testing

- ❖ In the testing phase the SW product is being validated – checked to comply with user needs.
 - ◆ Executed following already prepared testing plan that follows the system specifications / SRS
 - ◆ After correcting eventual discrepancies the testing phase needs to be repeated from the beginning.
- ❖ In the time of testing the documentation is finalized.

Testing ☺



(Release &) Deployment

- ❖ The product gets deployed to users
 - ◆ Users / customers with a help of developer test the whole system. Users / customers then accepts or rejects the system.
 - ◆ Testing can take place at the developer (FAT) or at the user (SAT).
- ❖ Deployment to the customer
 - ◆ The whole system is delivered to the customer and given into use.
 - ◆ User support may be provided in addition to user manuals.

Maintenance

- ❖ The system must fulfill user needs – correctly and constantly.
- ❖ Maintenance means changing of the product:
 - ◆ Corrective changes (error correction).
 - ◆ Adaptive changes (adjustments to modified conditions, e.g., extension of functionality, modification of interfaces...)
 - ◆ Perfective changes (making a product better, transfer to other platforms, additional functionalities)
 - ◆ Preventive changes (working against of software deterioration)
- ❖ The SW life cycle ends when SW is stopped being used.

Generic SW process phases

❖ Problem definition	Definition
❖ Feasibility study	Analysis
❖ Requirements analysis	
❖ System design	Design
❖ Component design	
❖ Implementation	Implementation
❖ Testing	
❖ Deployment	
❖ Maintenance	Maintenance

Models of a SW development process

- ❖ Model of a SW development process is a plan of executing phases in a software process.
- ❖ Each SW process includes all the phases, either implemented formally or informally and in different order.
 - ◆ The feasibility study cannot be executed without prior problem definition.
 - ◆ “Ad-hoc” approach means, that one or more phases are implemented informally.
 - ◆ SW design often shows up imperfections of the requirements specifications.
 - ◆ ...

Advantages of formal processes

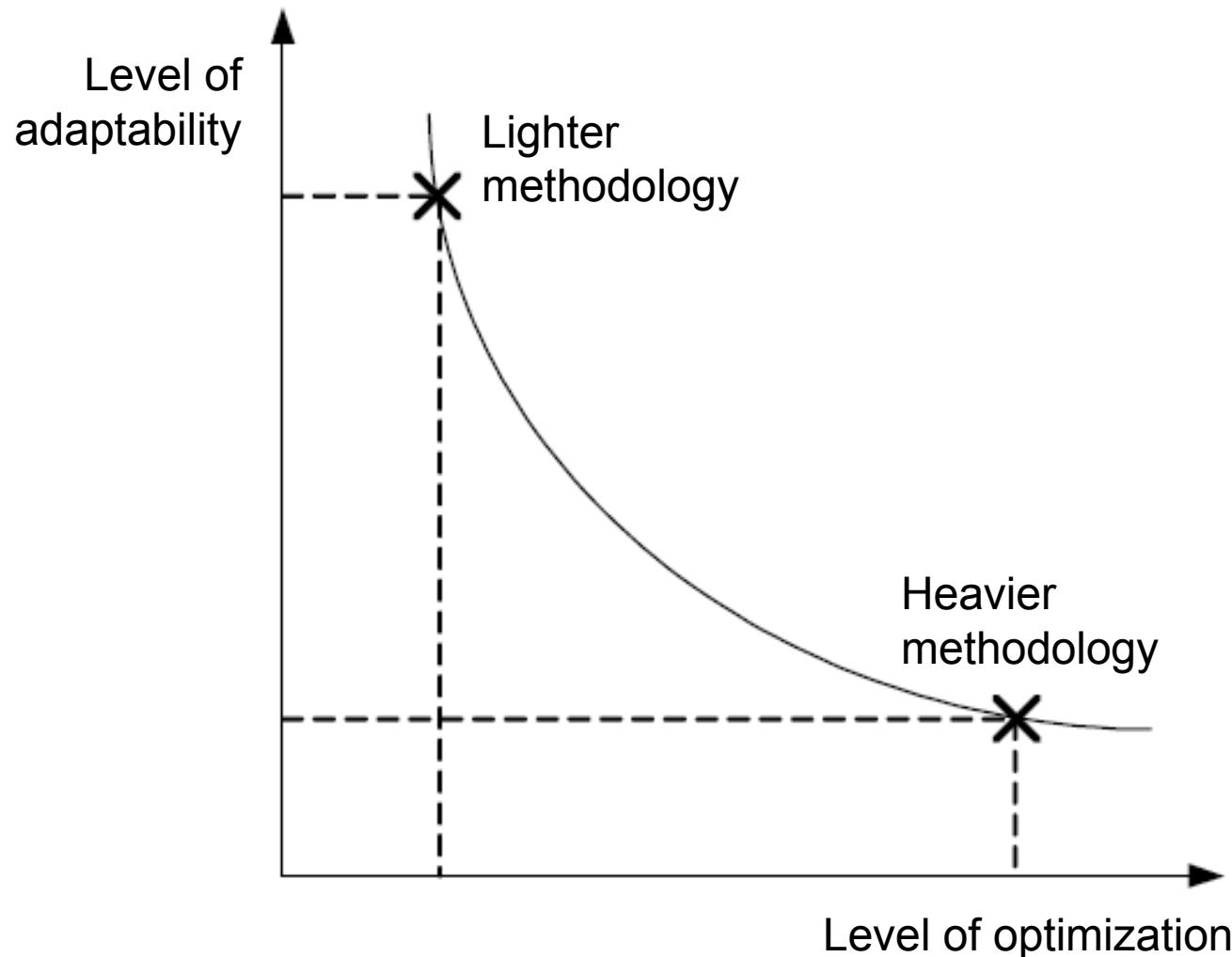
- ❖ Formal execution of SW process phases has several advantages:
 - ◆ Better control over resources.
 - ◆ Better relations with the customer.
 - ◆ Shorter development time.
 - ◆ Lower costs.
 - ◆ Higher quality and reliability.
 - ◆ Higher productivity.
 - ◆ Better work organization and coordination.
 - ◆ Less stressful work.

The larger the project the more expressed the advantages are.

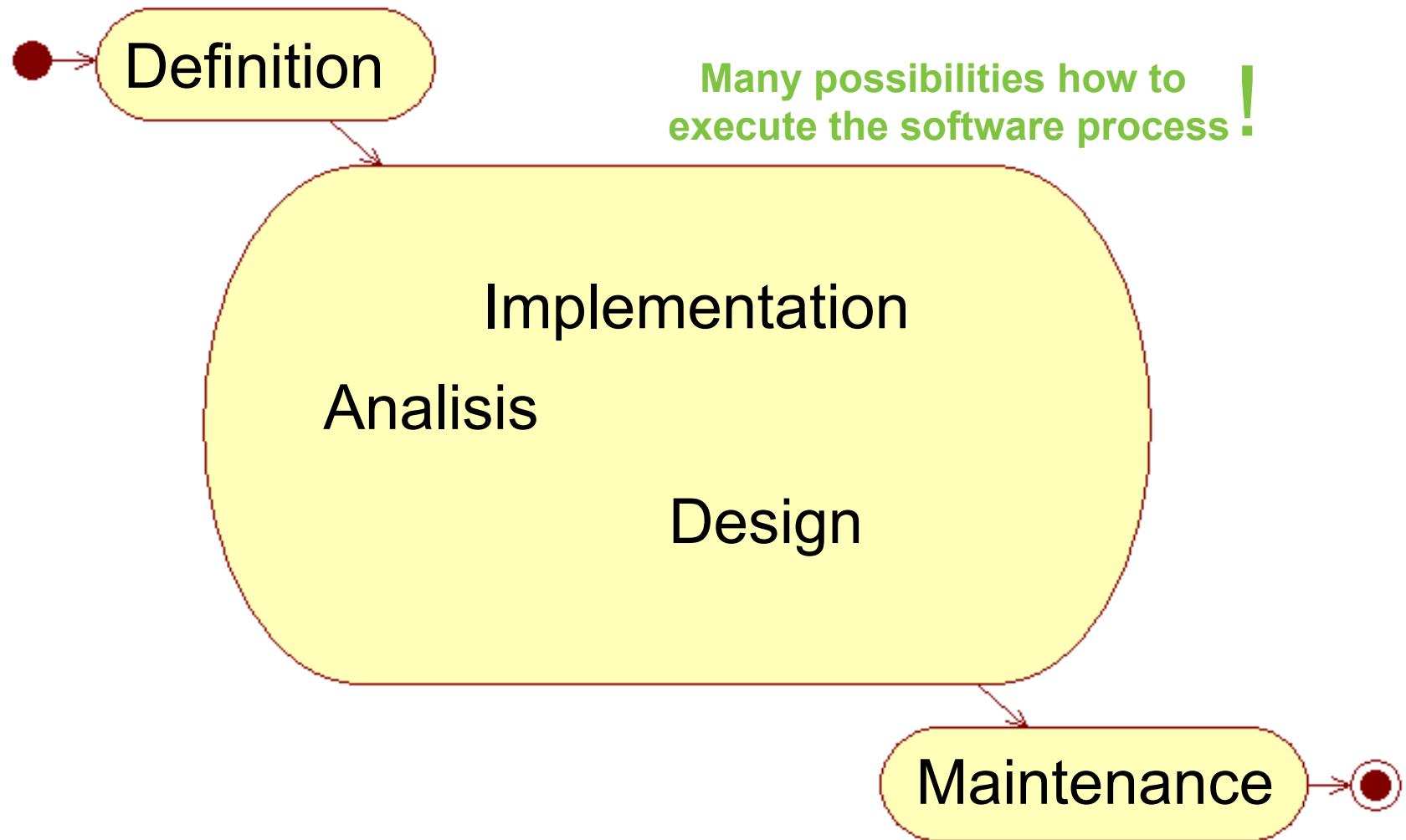
Advantages of informal processes?

- ❖ Advantages of formal processes are less expressed in smaller projects and the formalities are sometimes questioned:
 - ◆ Less documentation needs?
 - ◆ Less resources for organization and coordination?
 - ◆ Freedom and more motivated developers?
 - ◆ ...
- ❖ Informal implementation of phases can lead to problems in later phases!
 - if not sooner then during maintenance.

How heavy the methodologies are



SW development models



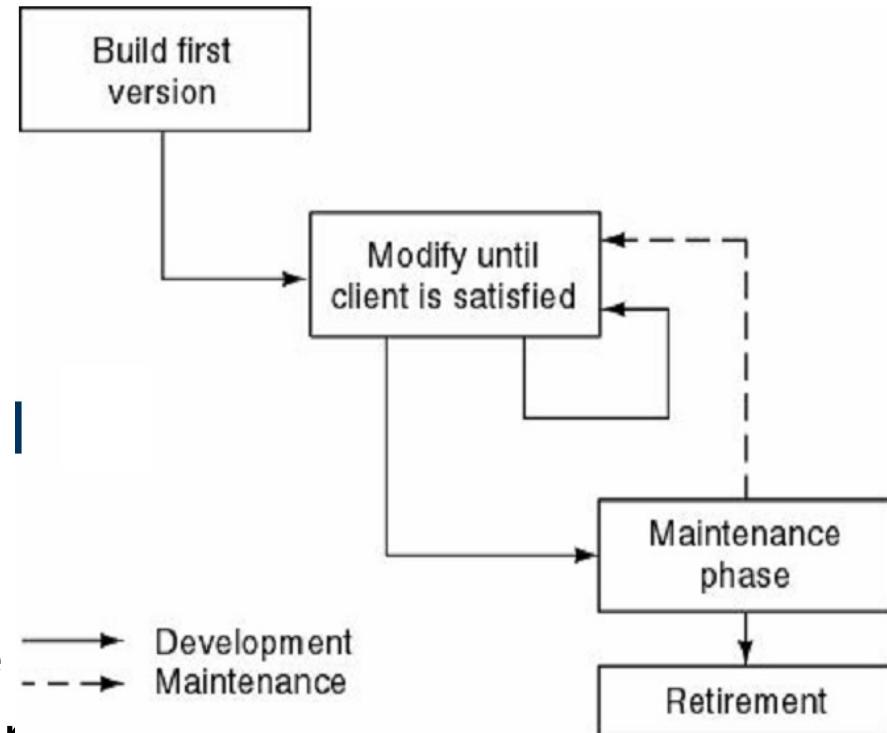
“Build and fix”

Difficulties

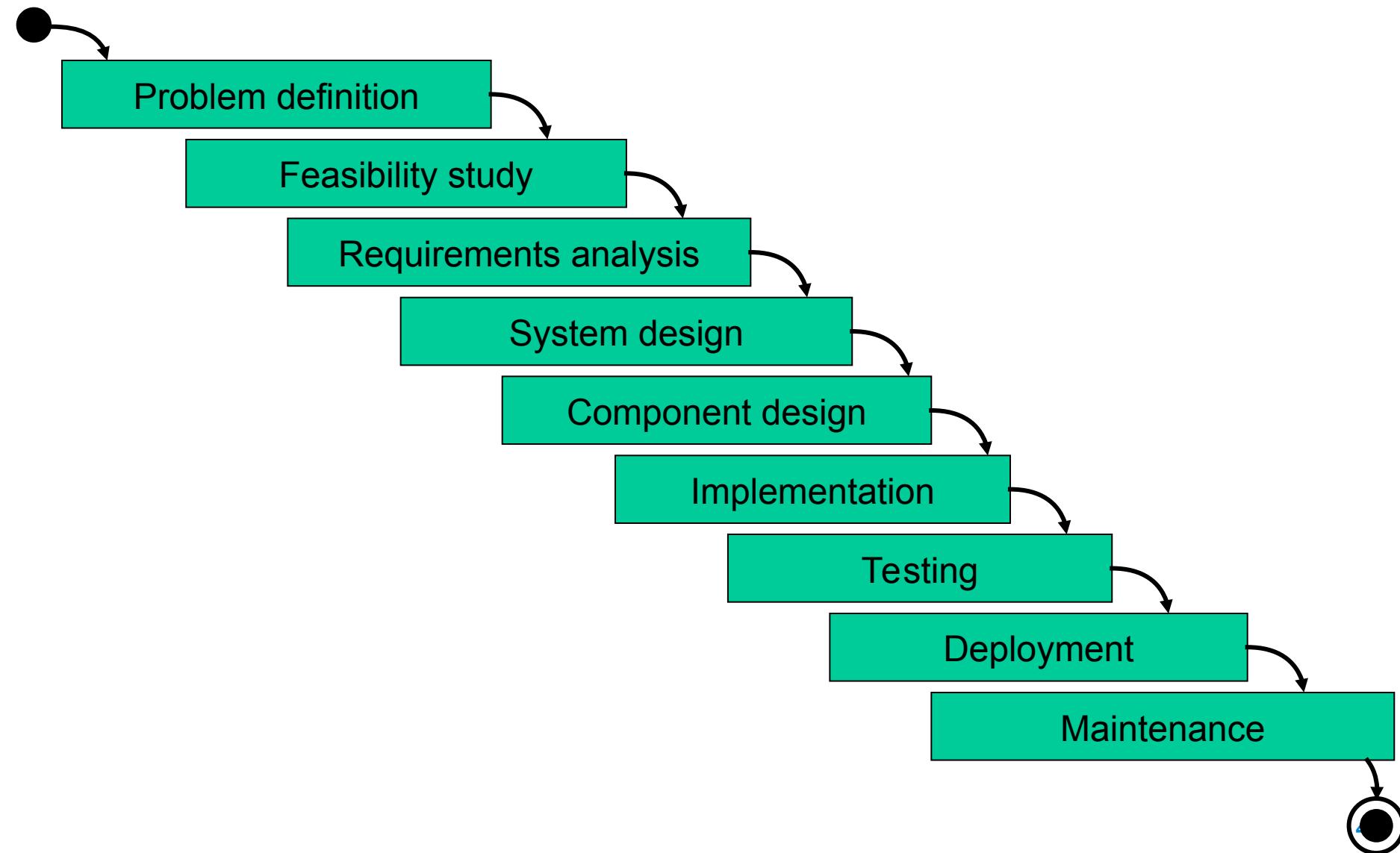
- ◆ Missing specification
 - ◆ Missing design plan
 - ◆ No defined phases
 - ◆ No milestones
 - ◆ ...

❖ Inconvenient approach!

- ◆ Impossible to control
 - ◆ Difficult quality assurance
 - ◆ High load on the customer, fall of trust in the product
 - ◆



Linear model 1/3



Linear model 2/3

- ❖ Linear (classic, cascade, waterfall) model:
 - ◆ Phases follow one after the other, without overlap.
 - ◆ Each phase ends with V&V
 - ◆ Possible only in the case of well defined requirements and project goals as later modifications are not possible.
 - ◆ Requires some patience from the customer. The working SW is available only after Deployment – at the end of the project.

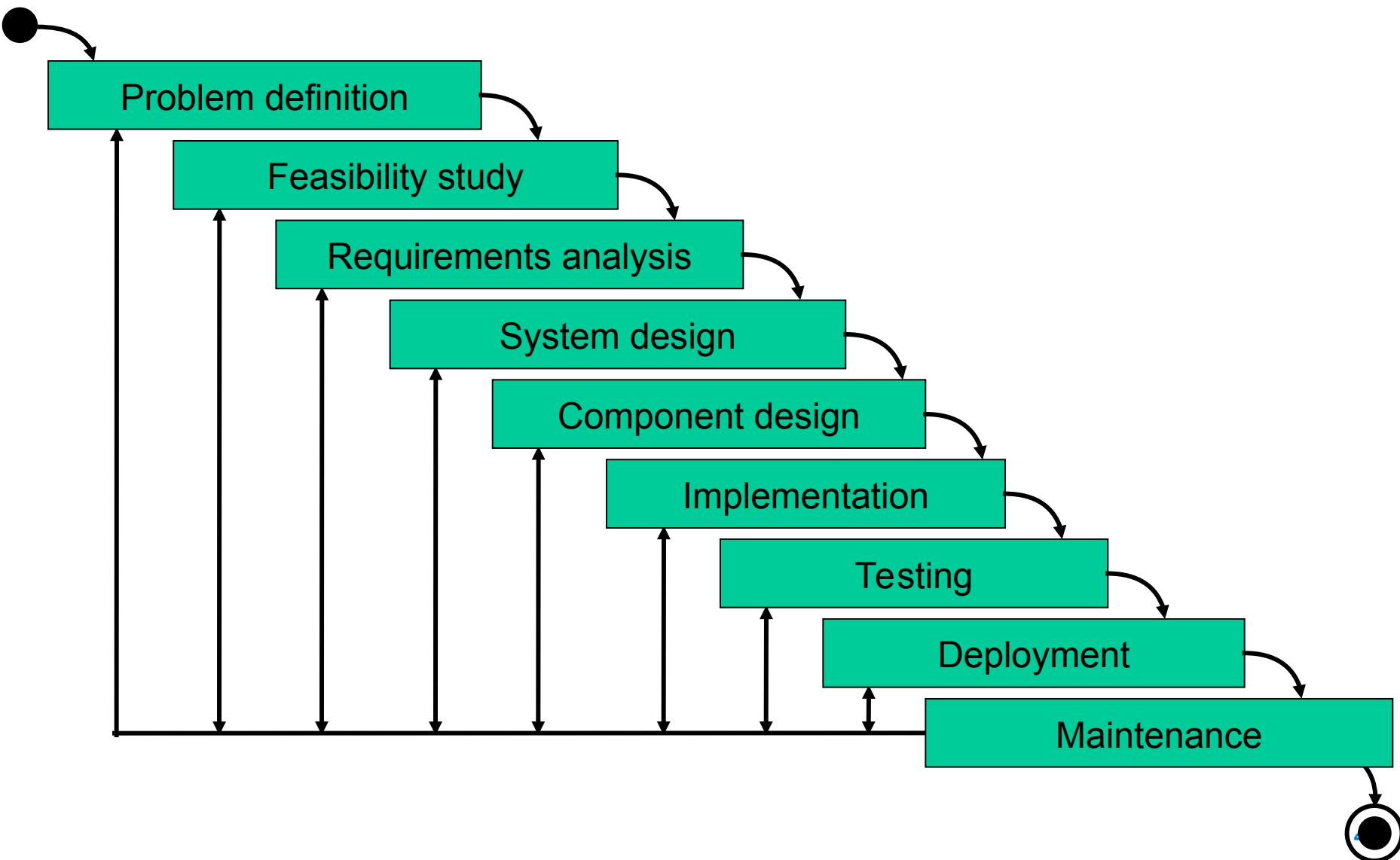
Linear model 3/3

- ❖ Advantages:
 - ◆ clarity,
 - ◆ Separation of phases,
 - ◆ Permanent quality control,
 - ◆ Good control over expenses.
- ❖ In practice only each next phase enable better understanding of the previous one, consequently revisions of previous phases are required. Linear model does not enable them!

Linear model with corrections

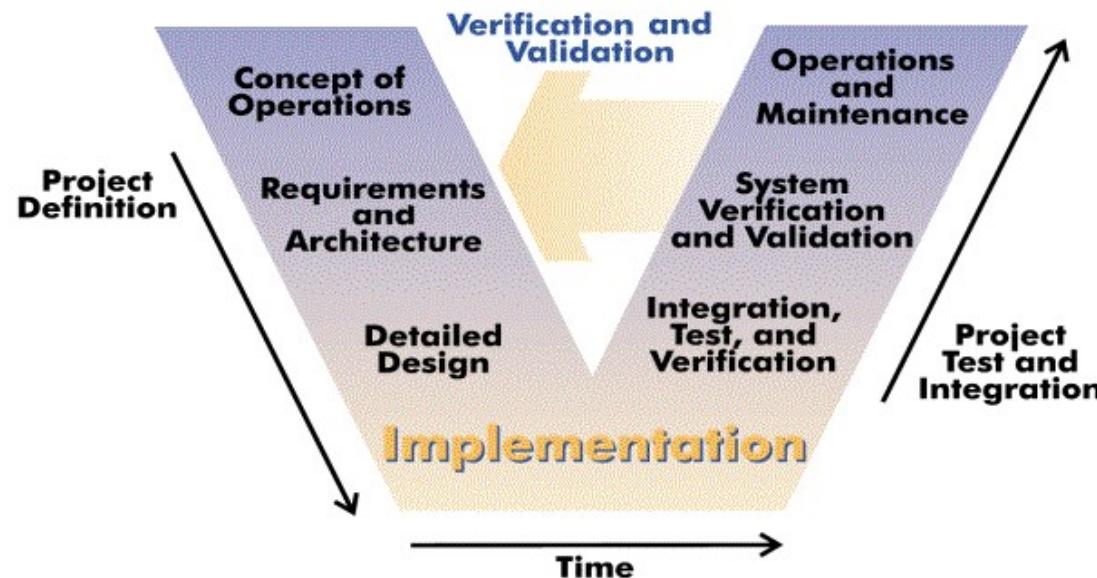
- ❖ Enables revisions of previous steps/phases.
- ❖ Possible partial overlap of phases.

Linear model with corrections

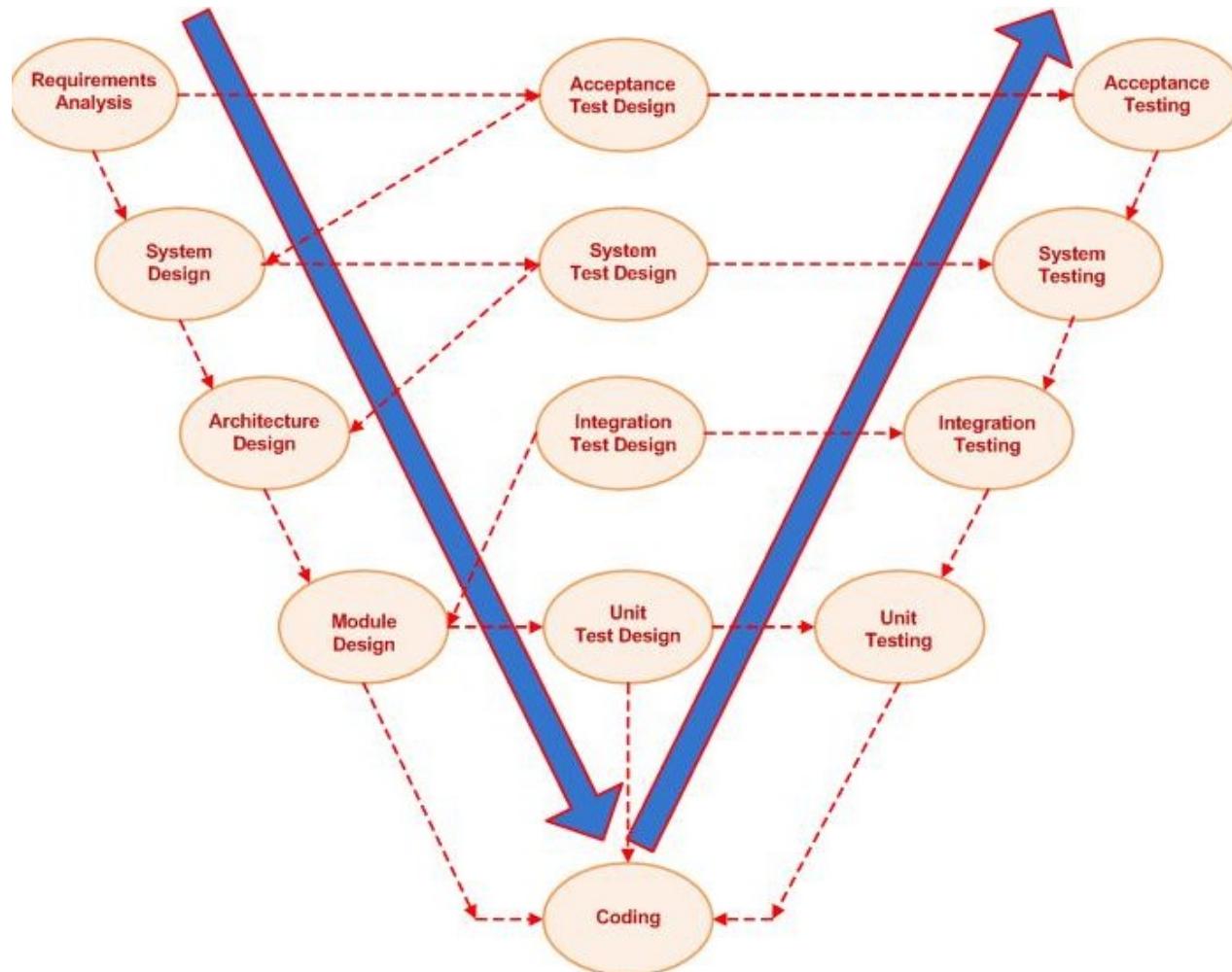


“V” model 1/2

- ❖ Extension of the linear model.
- ❖ Each phase of development is related to some phase of testing.

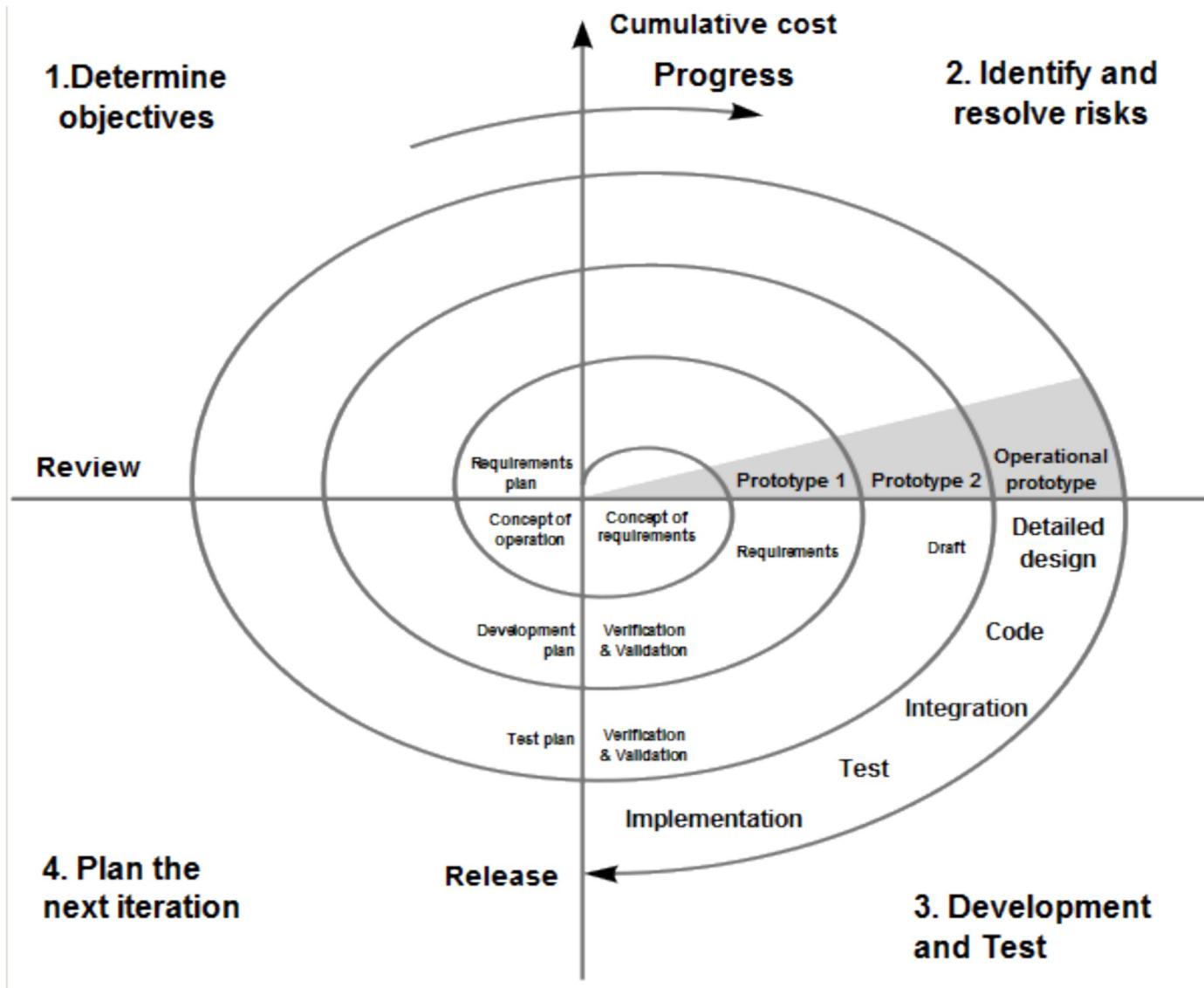


“V” model 2/2



Source: [http://en.wikipedia.org/wiki/V-Model_\(software_development\)](http://en.wikipedia.org/wiki/V-Model_(software_development))

Spiral model 1/9



Vir: http://en.wikipedia.org/wiki/Spiral_model

Spiral model 2/9

- ❖ Each cycle includes phases of the linear model cycle.
- ❖ In each cycle a prototype is developed.
- ❖ Each cycle includes four quadrants :
 - ◆ Definition of goals, alternatives, limitations.
 - ◆ Evaluation of alternatives, risk identification.
 - ◆ Development of a current cycle product.
 - ◆ Planning of the next cycle.

Spiral model 3/9

- ❖ Definition of goals, alternatives, limitations:
 - ◆ **Goals:** functionality, performance, interface definition, success factors...
 - ◆ **Alternatives:** new development, component reuse, buying a component or product, subcontracts...
 - ◆ **Limitations:** costs, time limits, interfaces...

Spiral model 4/9

- ❖ Evaluation of alternatives, risk identification:
 - ◆ Evaluation of alternatives according to goals and limitations.
 - ◆ Risk identification (lack of experience, new technologies, short time limits...)
 - ◆ Risk evaluation (failure probability, the sense of further development...)

Spiral model 5/9

- ❖ Development of a current cycle product:
 - ◆ The cycles of the linear model: design, implementation, testing.
 - ◆ Quality control (reviews, verification, validation)

Spiral model 6/9

- ❖ Planning of the next cycle:
 - ◆ Project planning of the next cycle
 - ◆ Configuration management (identification, organization and control of changes)
 - ◆ Test planning
 - ◆ Planing of system installations

Spiral model 7/9

❖ Advantages:

- ◆ Early identification of insurmountable obstacle – with reduced costs.
- ◆ Users get early insight into the system through prototypes.
- ◆ Development of critical components, first.
- ◆ Acceptable incomplete design and planning.
- ◆ Early feedback from users.

Spiral model 8/9

❖ Limitations:

- ◆ Loss of time due to redesigns, redefinition of requirements, risk analysis and building of prototypes.
- ◆ A complex model that does not define the end of a project – number of cycles.
- ◆ Developers are not active (efficiently used) in all of the quadrants.

Spiral model 9/9

❖ When to use it?

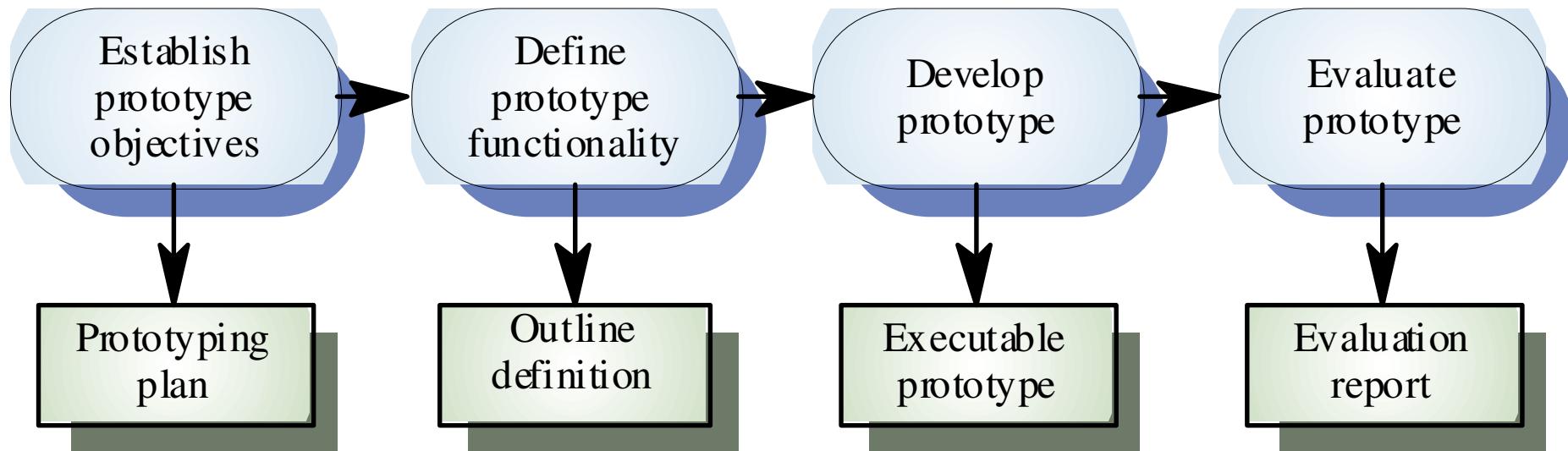
- ◆ When prototypes are acceptable.
- ◆ When risk management is crucial. Projects with higher risks.
- ◆ When longer development time period is acceptable.
- ◆ In the case of complex requirements.
- ◆ When changes of requirements are expected.

Using prototypes 1

- ❖ Using prototypes for SW development is a way of rapid development.
- ❖ Original purpose of prototypes was to help customer and developers to better understand requirements.
 - ◆ Users can check how their requirements are reflected in the system.
 - ◆ Prototype reveals error and missing requirements.
- ❖ Prototype development is an activity of reducing risks at requirements analysis.
- ❖ Prototype is available early and can be used to educate users and test systems.

Prototype development 2

❖ Prototype development



Vir: Ian Sommerville, Software Engineering, 6th edition. Chapter 8, 2000.

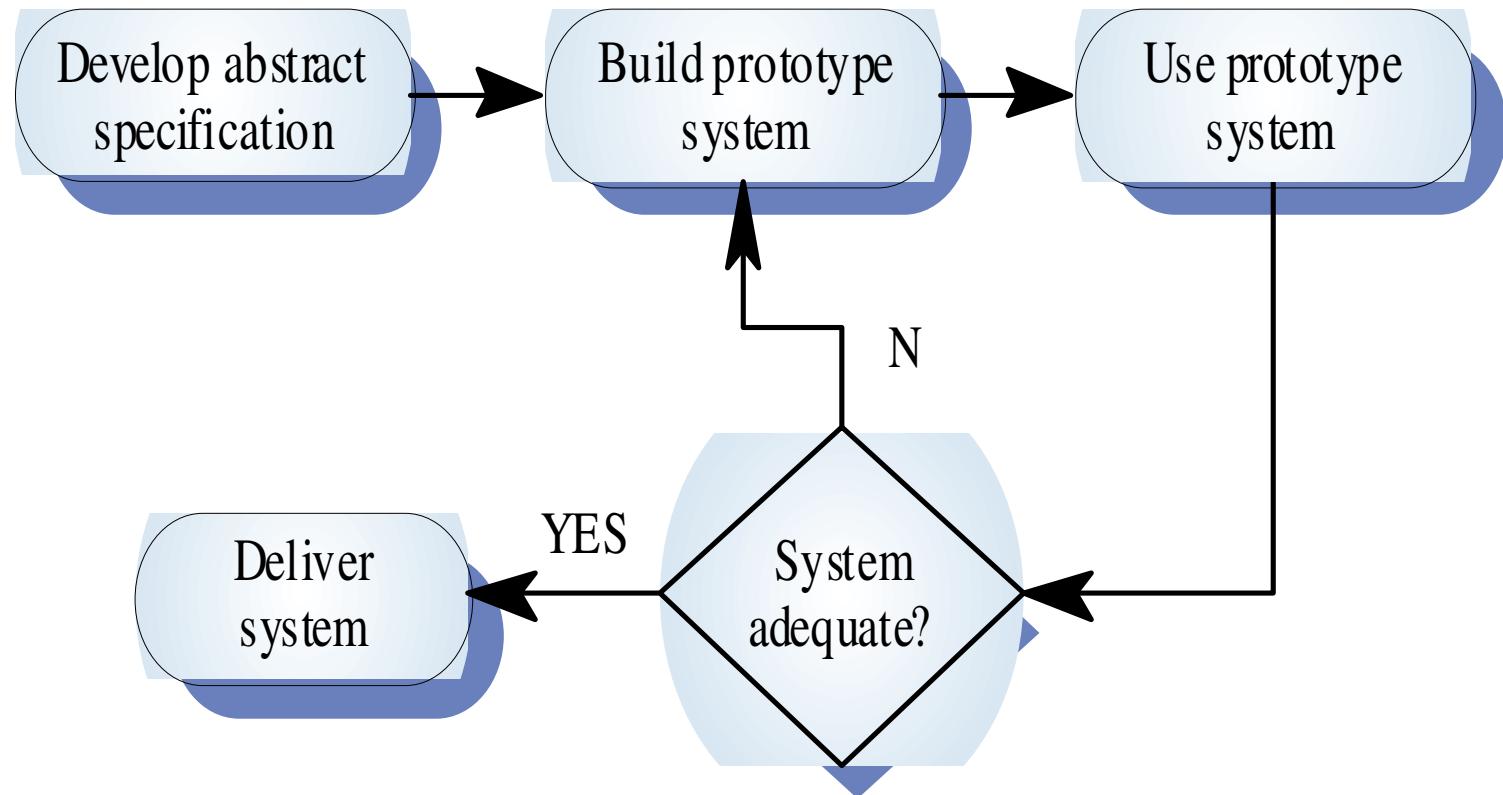
Prototype development 3

- ❖ Historically a prototype was considered inferior and so the prototype development was always followed by further development.
Today we have two options:
 - ◆ Evolutionary development
 - An initial prototype is gradually improved and leads to the final SW system.
 - The purpose of the prototype is to build the final system.
 - Development starts at the most clear requirements.
 - ◆ Development with prototype omission
 - Prototype serves only for clarifying requirements and is later abandoned. The final system is developed independently, based on clarified requirements.
 - Development starts with most unclear requirements.

Evolutionary development 1/4

- ❖ Evolution development is needed when specifications cannot be in advance to developing the system (prototype), e.g., research, artificial intelligence...
- ❖ Verification is not possible, as there are no specifications. Validation can only be made as a demonstration of system capabilities.
- ❖ Evolution development is based on rapid prototyping techniques.

Evolutionary development 2/4



Src.: Ian Sommerville, Software Engineering, 6th edition. Chapter 8, 2000.

Evolutionary development 3/4

- ❖ Prototype can under certain circumstances serve as system specifications, but not always and completely:
 - ◆ Some requirements cannot be included in prototypes (security critical functions)
 - ◆ Prototypes legally cannot be considered as a contract agreement.
 - ◆ Nonfunctional requirements cannot be verified on prototypes (security, testability, maintainability, extendability, upgradability...)

Evolutionary development 4/4

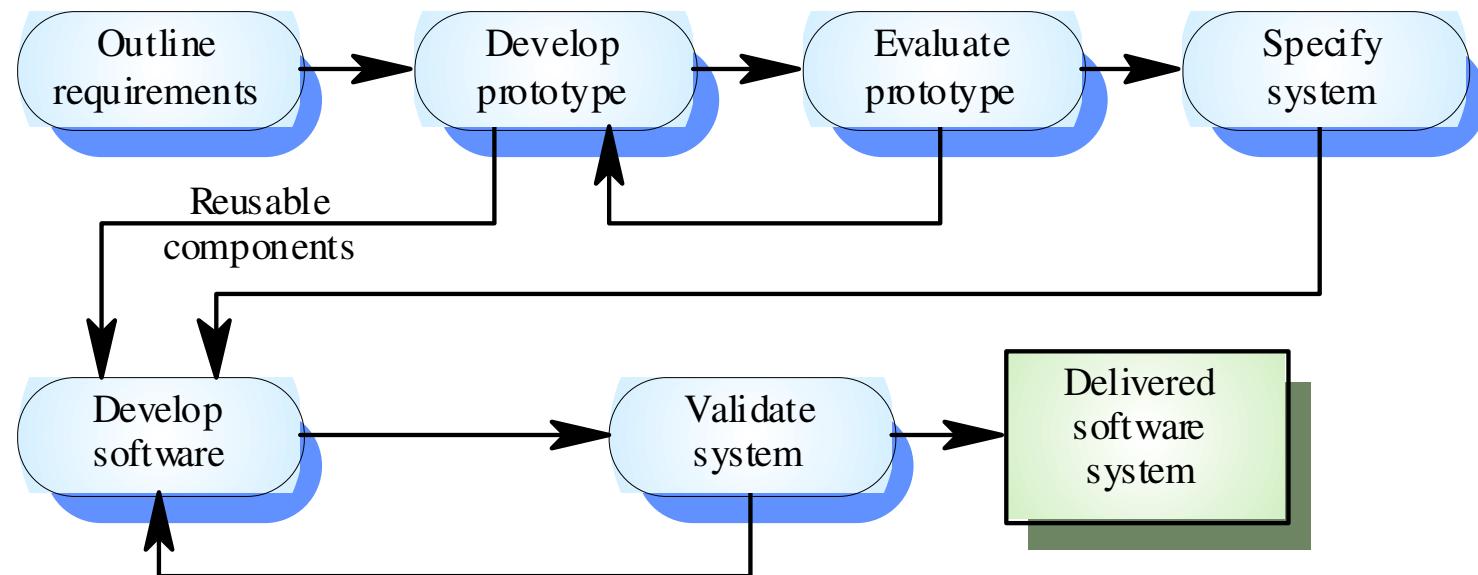
❖ Advantages and disadvantages:

- ◆ + Early delivery. Sometimes early delivery is more important than the functionality or long term maintenance.
- ◆ + User inclusiveness increase the probability that the system satisfies user needs and increases probability of the system being used.
- ◆ - Project management typically expects linear models.
- ◆ - Prototype development without specifications requires additional developer abilities.
- ◆ - Maintenance difficulties. Constant changes have negative influence on the system structure, making long term maintenance difficult.
- ◆ - Contractual difficulties. Users/customers must accept additional obligations, multiple deliveries for deployment.

Throwaway prototyping 1/3

- ❖ The purpose of a prototyping is to reduce risks at requirements analysis.
- ❖ A prototype is developed based on initial specifications for experimenting (different properties, technologies...) and is then abandoned.
- ❖ A prototype is not meant to be used in/for the final system!
 - ◆ Prototypes are not developed considering all the required system properties.
 - ◆ Prototypes are not documented and would be difficult to maintain.
 - ◆ Architecture of a prototype is developed “ad-hoc” and thus less appropriate for long term maintenance.

Throwaway prototyping 2/3



Src.: Ian Sommerville, Software Engineering, 6th edition. Chapter 8, 2000.

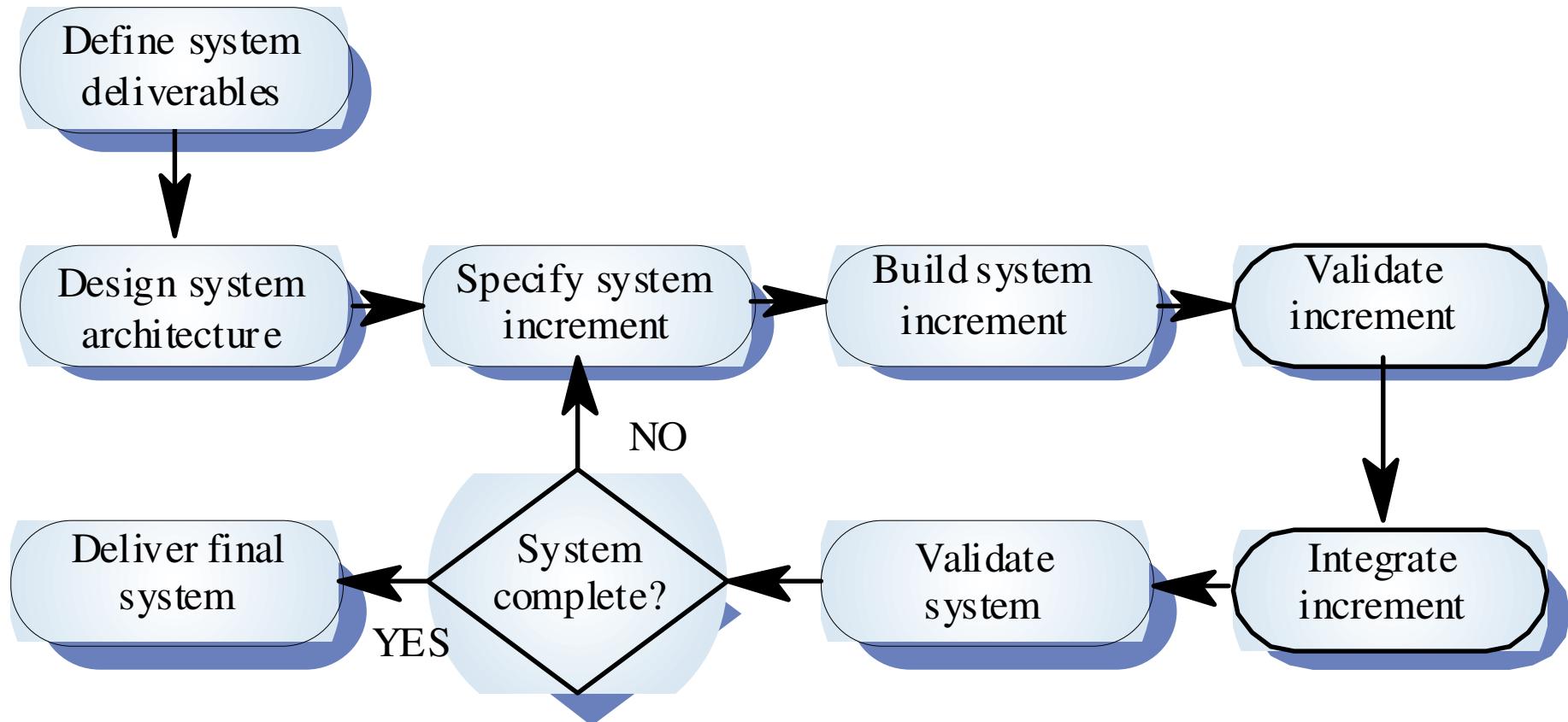
Throwaway prototyping 3/3

- ❖ Developers are often under pressure to use a prototype as a final system.
- ❖ Why is this not acceptable?
 - ◆ Not satisfying all requirements, especially not nonfunctional ones.
 - ◆ Lack of documentation.
 - ◆ Architecture difficult to maintain.
 - ◆ Quality standards are usually not considered when building prototypes.

Phased development 1/3

- ❖ System developed and delivered in multiple phases.
- ❖ The basis for development of the first phase is an architectural design of the whole system.
- ❖ Phases deployed to users already enable some incomplete functionality although some functionality is missing.
- ❖ Users can check or even use functionality of some phases while others are still being developed.
- ❖ Has advantages over prototyping (evolution process) as it assures better project clarity, control, management and better final architectural design.

Phased development 2/3



Src.: Ian Sommerville, Software Engineering, 6th edition. Chapter 8, 2000.

Phased development 3/3

- ❖ The initial system with limited functionality can be deployed sooner and already used enabling quick return of investment.
- ❖ Development of further phases is based on experience obtained from previous phases. (better understanding of requirements).
- ❖ Examples?

Unified Process

- ❖ Unified Process (UP) is a modern process model framework that can be adapted to meet needs of specific organizations or projects.
- ❖ IBM acquired UP and offers it under the name Rational Unified Process (RUP).
- ❖ The framework defines development phases and engineering disciplines to perform these phases.
- ❖ Each phase is built of steps, which are the basic elements of a process.

Unified Process

- ❖ A project consists of four phases:
 - **Inception** Establish the business case for the system.
Ends with the Lifecycle Objective Milestone
 - **Elaboration** Develop an understanding of the problem domain and the system architecture.
Ends with the Lifecycle Architecture Milestone
 - **Construction** System design, programming and testing.
Ends with the Initial Operational Capabilit Milestone
 - **Transition** Deploy the system in its operating environment.
Ends with the Product Release Milestone

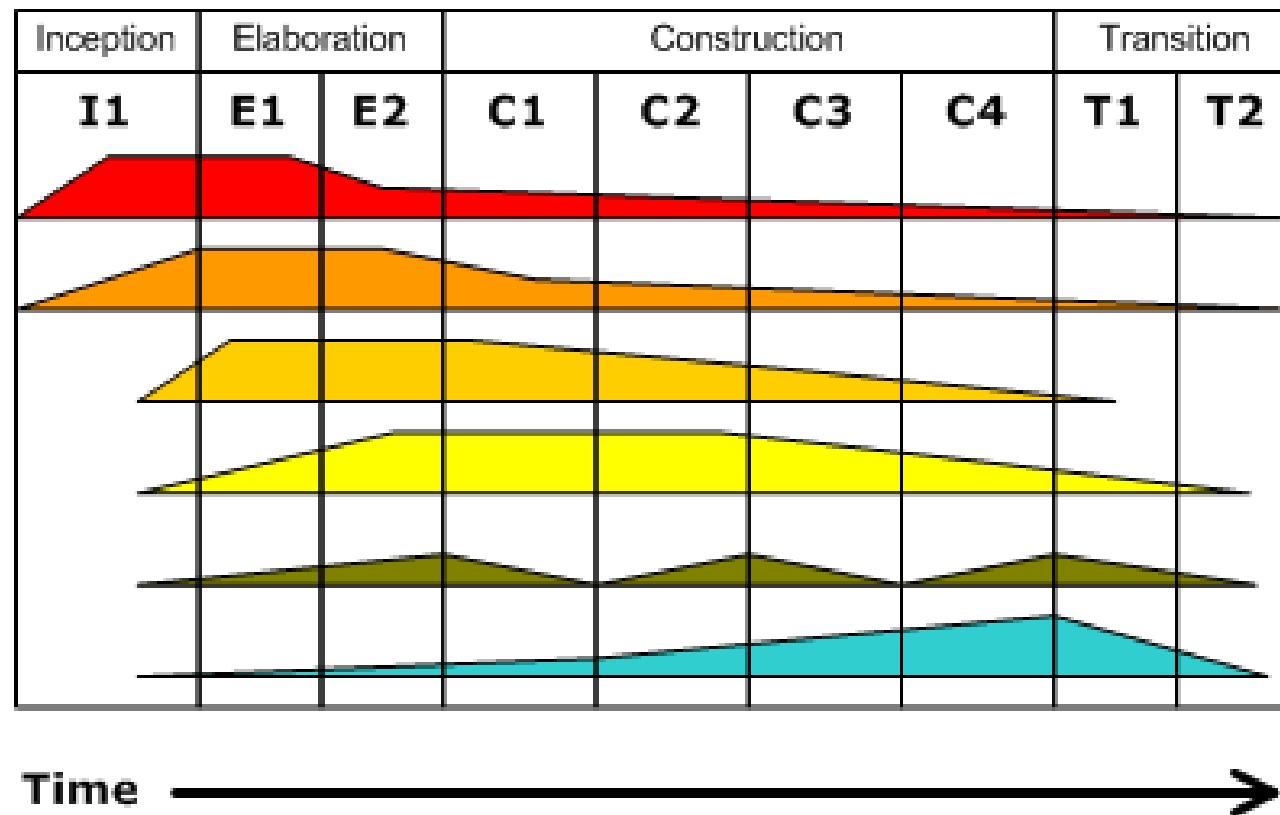
Unified Process

- ❖ Unified Process defines six engineering disciplines:
 - ◆ business modeling
 - ◆ requirements discipline
 - ◆ analysis and design
 - ◆ implementation
 - ◆ test discipline
 - ◆ deployment

Unified Process

Iterative Development

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



Unified Process

- ❖ Process built from basic building blocks defining what needs to be made, which competences are needed and how to reach the goal:
 - ◆ **Roles (who):** knowledge, competences, responsibilities.
 - ◆ **Products (what):** the result of a task - documents, models, data, programs.
 - ◆ **Task (how):** description of a process to reach the goal.

Unified Process

❖ UP good practice:

- Develop software iteratively
- Manage requirements
- Use component-based architectures
- Visually model software
- Verify software quality
- Control changes to software

Software process frameworks

- ❖ An organization can build its own software process framework to define its processes.
- ❖ Each process consists of steps, which:
 - ◆ Define a phase,
 - ◆ Require some engineering discipline,
 - ◆ Define roles of developers involved in the process,
 - ◆ Result in predefined products,
 - ◆ Are supplemented with guidelines to efficiently execute the process.
- ❖ Software process frameworks (freeware example):
 - ◆ Eclipse Process Framework Project (EPF)
<http://www.eclipse.org/epf/>

Software process maturity levels

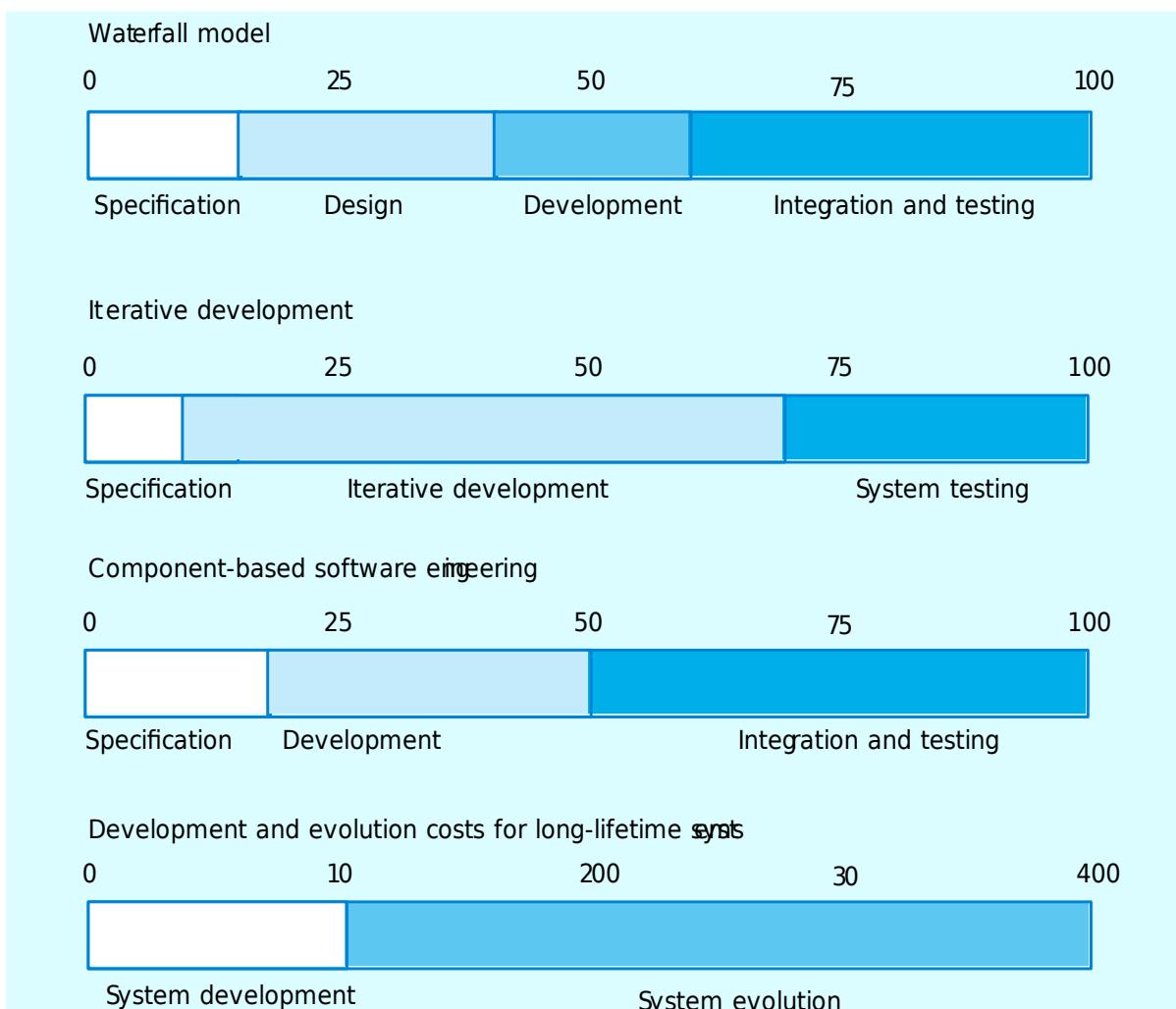
A measure of maturity of software processes in an organization, also known as Capability maturity model (CMM):

1. **Initial** – Processes depend solely on individuals
2. **Repeatable** - established basic project management policies. Experiences may be used for similar projects.
3. **Defined** – documentation of the standard guidelines and procedures takes place. Each project follows predefined processes.
4. **Managed** – quantitative quality goals are set for the organization for software products as well as software processes.
5. **Optimizing** – continuous process improvement in the organization using quantitative feedback.

Software costs

- ❖ Approx. 60% of costs is development and 40% testing.
For custom built software costs of modifications after the first deployment exceed the initial development costs.
- ❖ Costs depend on the system type and requirements as performance and reliability.
- ❖ Distribution of costs depend on a software process model used.

Cost distribution



Software engineering

Diagramming techniques

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Why diagrams?

- ❖ Clear description of models.
- ❖ The basic tool for modeling:
 - ◆ Enable description of complex systems
 - ◆ Unambiguous presentation contributes to easier understanding of a problem and a solution.
 - ◆ Make modeling of system structure and behavior easier.

Diagrams and models

- ❖ Diagram is a **partial** graphic presentation of a system model.
- ❖ Model can include multiple diagrams and documentation.
 - ◆ Example: the use-case model consists of a use-case diagram and a document describing the use-case.
- ❖ This lecture is limited only to diagramming techniques (diagrams) and NOT the overall models.

Diagrams for SW product development

- ❖ Flowchart
- ❖ Data Flow Diagram, DFD
- ❖ Entity Relationship Diagram, ER diagram
- ❖ UML diagrams (Unified Modeling Language) – diagramming technique for various diagram types, specified for SW engineering.
- ❖ Others (depending on the needs)

Diagram (and model) types

❖ **Context**

- ◆ Description of a system from outside – in his environment.

❖ **Behavioral**

- ◆ System behavior: use-cases, process behavior, data flow, state transitions...

❖ **Structural**

- ◆ Structure of a system, components, objects, data...

Diagramming (model) domains

❖ Application domain

- ◆ For requirement analysis
- ◆ System environment

❖ Solution domain

- ◆ System design, component/object design
- ◆ System development technologies

Model levels

- ❖ Conceptual model
 - ◆ Basic system properties, high level, broad/coarse view on the system.
- ❖ Logical model
 - ◆ Supplements the conceptual model with details, limited to system operation and usage (processes, data...).
- ❖ Physical model
 - ◆ Adds implementation details to the logical model (types, variables, functions...)

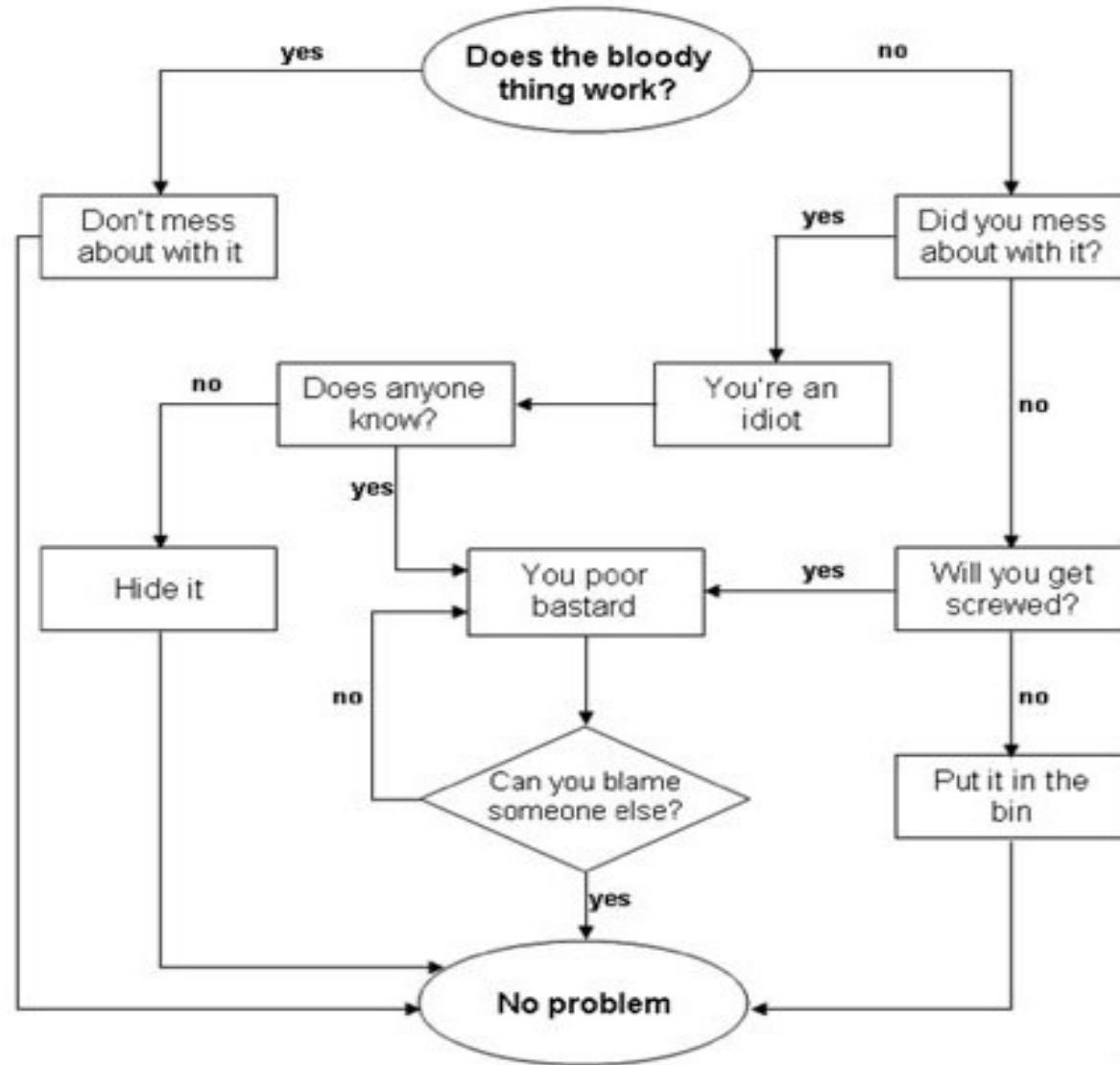
Diagramming recommendations

- ❖ Avoid line crossing.
- ❖ In the case of line crossing use jump instead.
- ❖ Avoid diagonal and curved lines.
- ❖ Elements shall have comparable size.
- ❖ Show only what is needed.
- ❖ Use renowned notations whenever possible.
- ❖ Reorganize large diagrams in multiple smaller ones.
- ❖ Focus on the contents first, then on the appearance.
- ❖ Organize diagrams from left to right and from top to bottom.
- ❖ Use suitable names (considering the meaning and usage area)
- ❖ Mark unknowns with comments and question marks.
- ❖ Color usage.

Flowchart

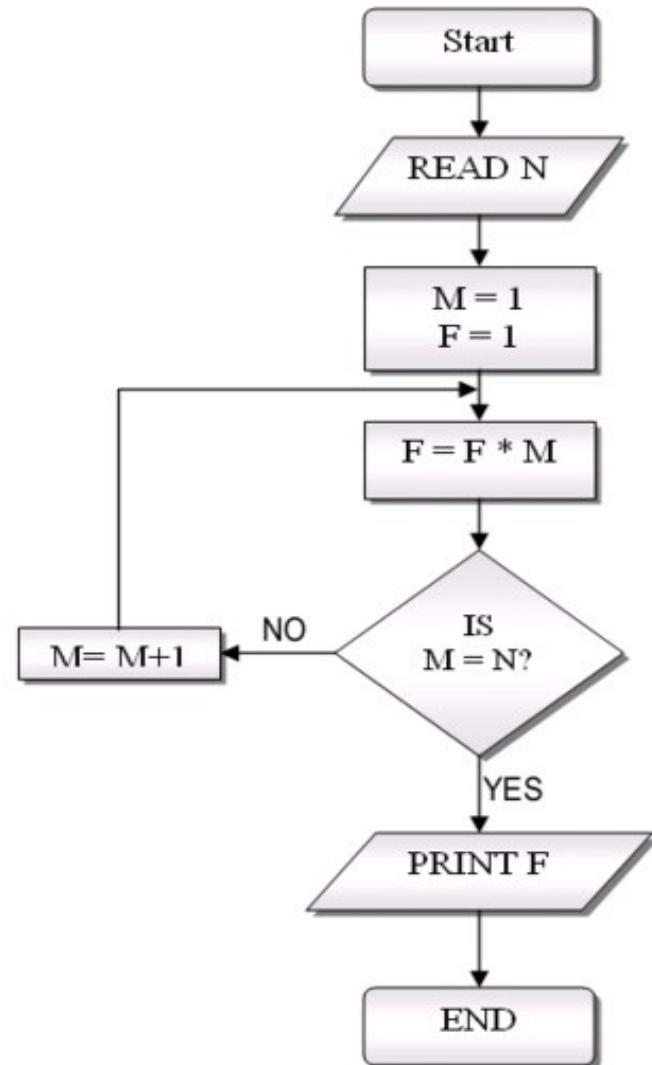
doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Flowchart 😊



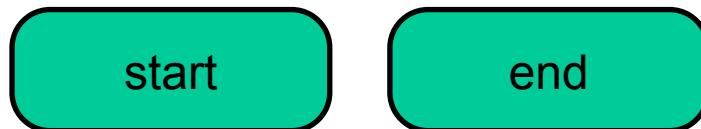
Flowchart

- ❖ Example: flowchart for computing factorial ($N!$).
- ❖ Diagram consists of symbols for start, end, control flow, process steps, I/O operations, decisions...



Flowchart symbols

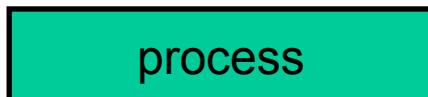
- ❖ Start, end: rounded rectangles or circles



- ❖ Control flow: arrows



- ❖ Process steps: rectangles

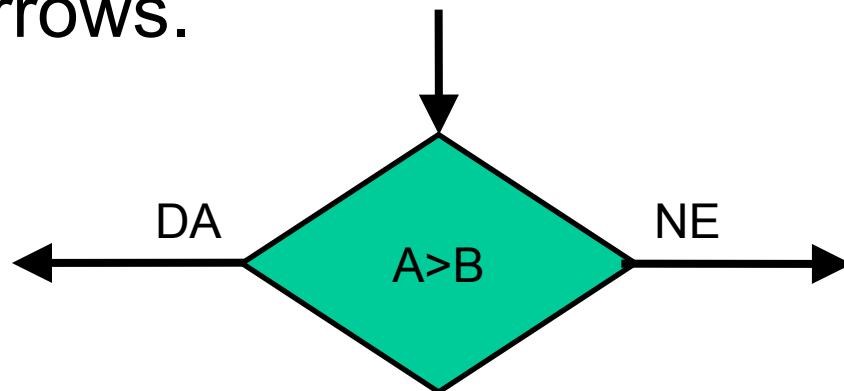


Flowchart symbols (2)

- ❖ I/O operations: parallelograms



- ❖ Conditions, decisions: rhomboids, labels at exiting arrows.



DFD, Data Flow Diagram

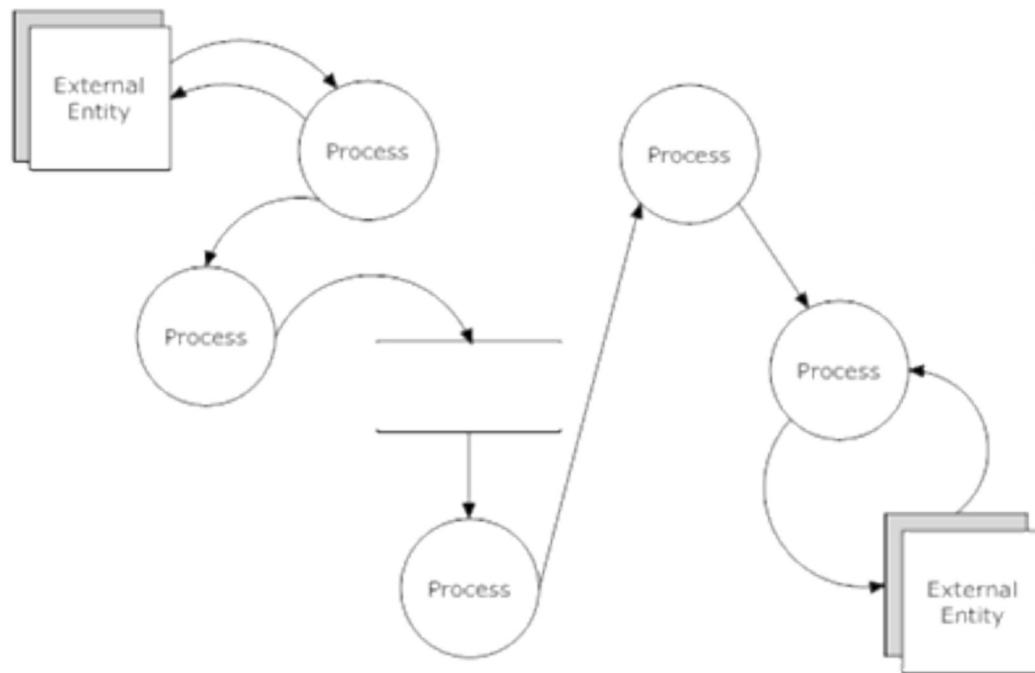
doc. dr. Peter Rogelj (peter.rogelj@upr.si)

DFD

- ❖ Modeling of data flow, not control flow.
- ❖ Logical view on a system, not physical.
- ❖ Convenient for communication with users, management and information system experts.
- ❖ Suitable for analysis of existing and proposed systems.
- ❖ Relatively simple technique.

DFD

- ❖ Shows how data is processed in the sense of inputs, processes and outputs.
- ❖ Remember from Systems III – Information systems.



This diagram has several flaws. Can you find them?

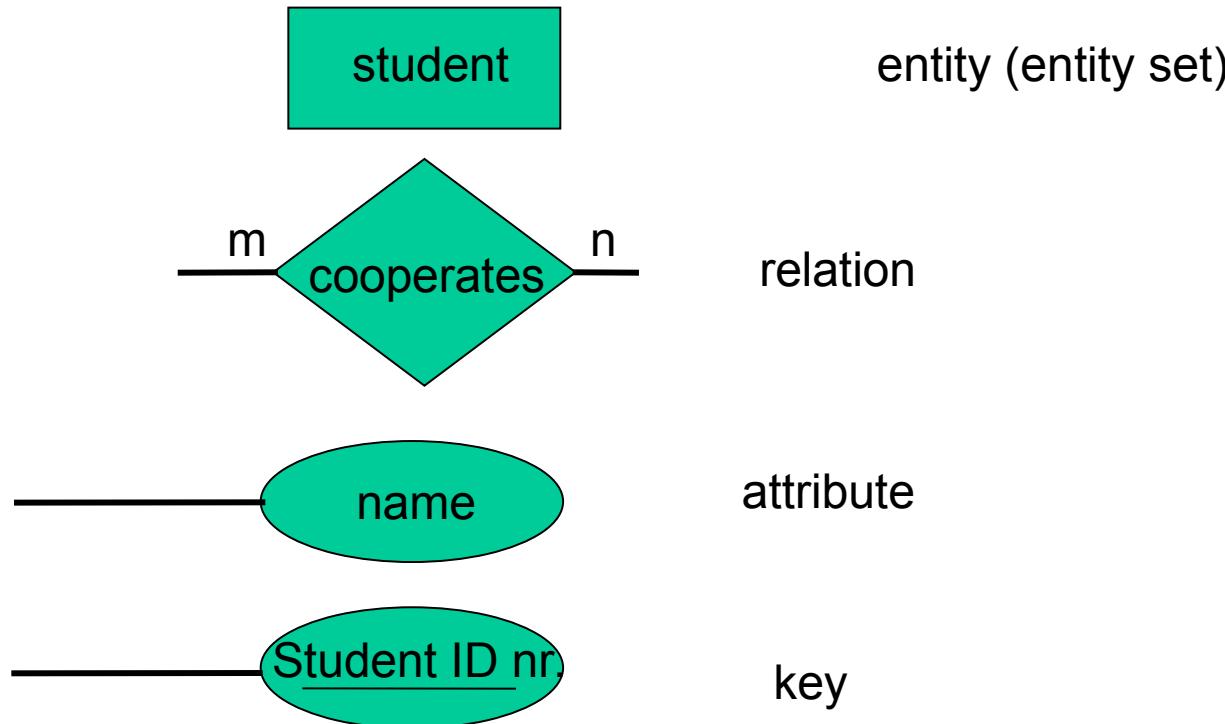
ERD, Entity Relationship Diagram

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

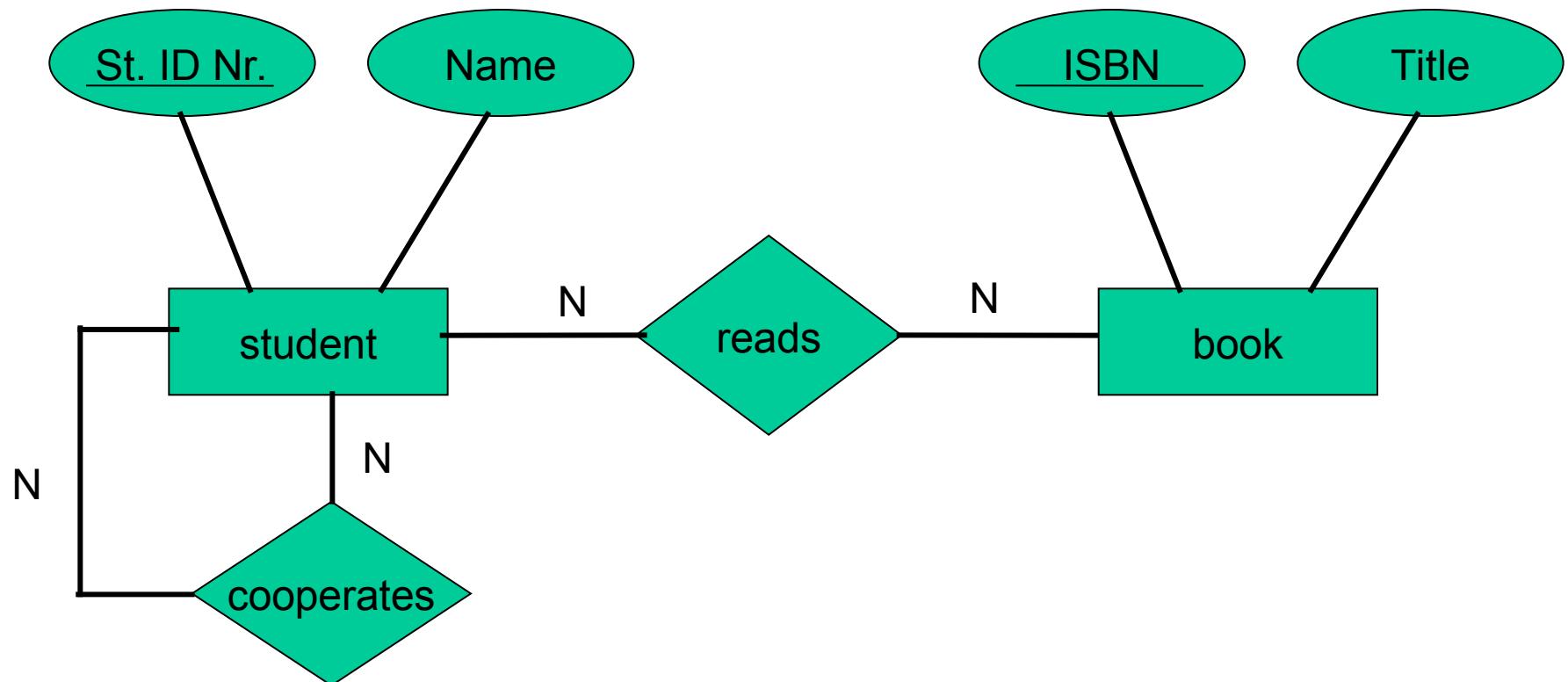
ERD

- ❖ Intended for illustration of a data structure.
- ❖ Definitions:
 - ◆ **Entity** – a real world object, which can be distinguished from other objects. Entity consists of a set of attributes.
 - ◆ **Entity set** – a set of similar entities, with the same set of attributes.
 - ◆ **Attribute** – a characteristic of an entity.
 - ◆ **Relationship** – relation between two or more entities
 - ◆ **Key** – an attribute that enables differentiation between entities.

Symbols:



Structure of ER diagrams



Cardinality

(0,1)



Optional one

(1,1)



Mandatory one

(0,N)



Optional many

(1,N)



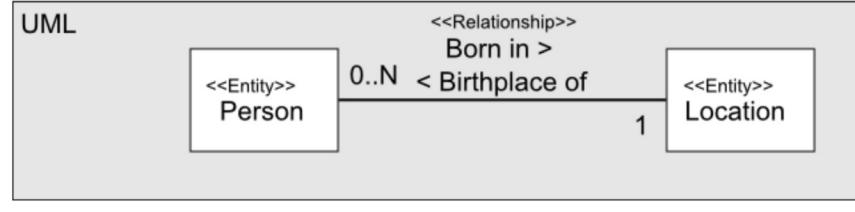
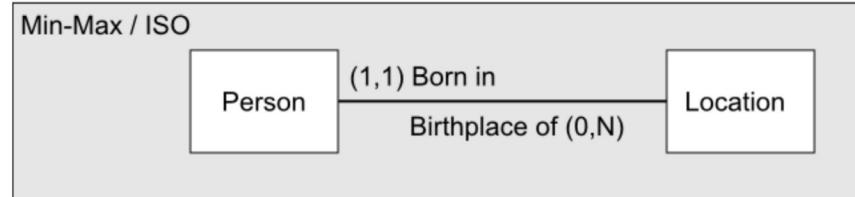
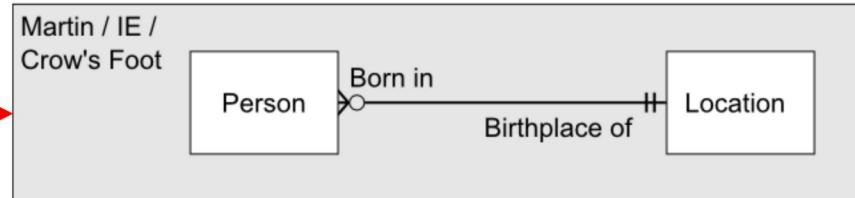
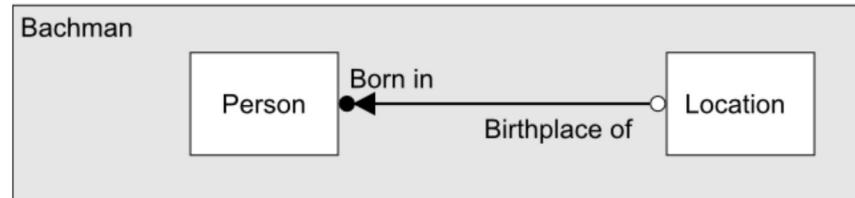
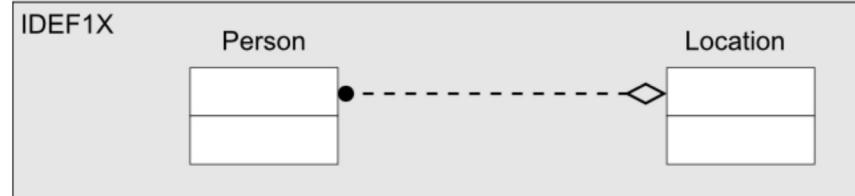
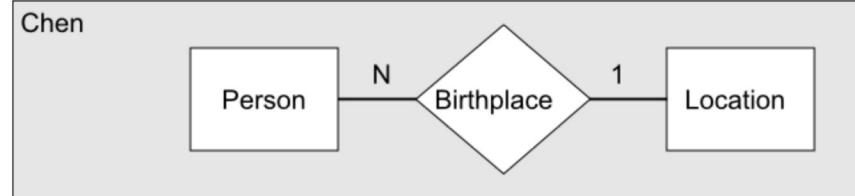
Mandatory many

Notations

- ❖ Multiple notations in use
- ❖ No official standard defined.

EMRIS

Unified methodology for the development of information systems

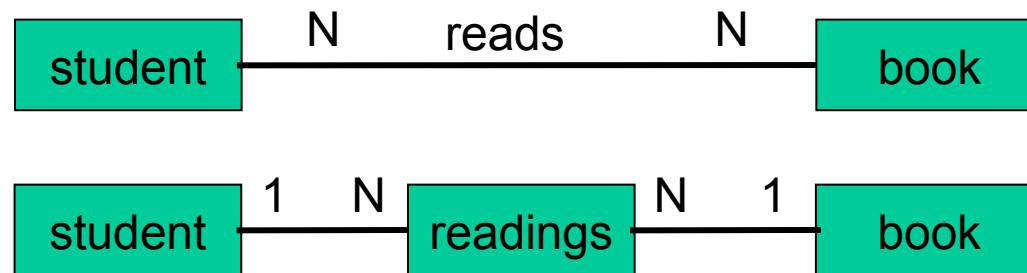


Src.:

http://en.wikipedia.org/wiki/Entity_relationship_diagram

Mapping from ER to relational model

- ❖ ER diagram is mapped to relational model, which defines how data is stored in a relational database.
- ❖ Example:
 - ◆ Cardinality N:N mapped using two tables for two entity types and an additional for relation.



Tools

- ❖ **SQL power architect**

- ◆ <http://www.sqlpower.ca/page/architect>

- ❖ **Mysql Workbench**

- ◆ <https://www.mysql.com/products/workbench/>

- ❖ **Toad data modeler**

- ◆ <http://www.quest.com/Toad-Data-Modeler/>

- ❖ **Others...**

UML diagrams

(the Unified Modeling Language)

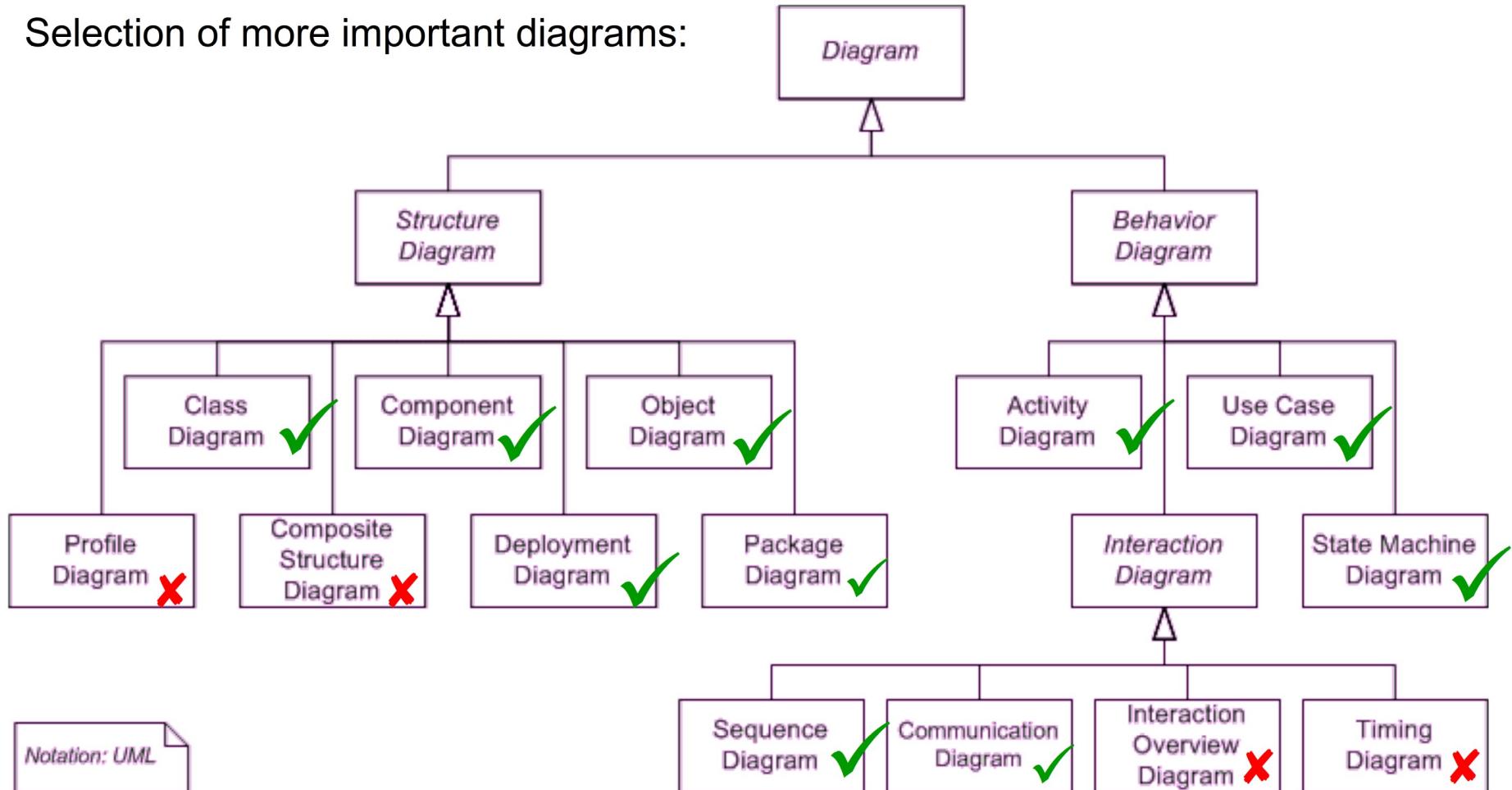
doc. dr. Peter Rogelj (peter.rogelj@upr.si)

UML diagrams

- ❖ UML (The Unified Modeling Language) is used for specification, visualization, modification, construction and documentation of processes and objects during system development.
- ❖ The focus is on object oriented approach.
- ❖ It is a set of standardized system representations.
- ❖ For 80% of all software only 20% of the UML is needed.

UML diagrams

Selection of more important diagrams:



Structure diagrams

- ❖ **Class diagram** depicts classes, alongside with their attributes and their behaviors .
- ❖ **Component diagram** breaks down the system into smaller components to depict the system architecture.
- ❖ **Composite structure diagram** represents the internal structure of a class and the relations between different class components.
- ❖ **Deployment diagram** is used to visualize the relation between software and hardware.
- ❖ **Object diagram** depicts instances (objects) of the classes present at certain time.
- ❖ **Package diagram** shows how a system is divided into logical units (packages) including their relations.

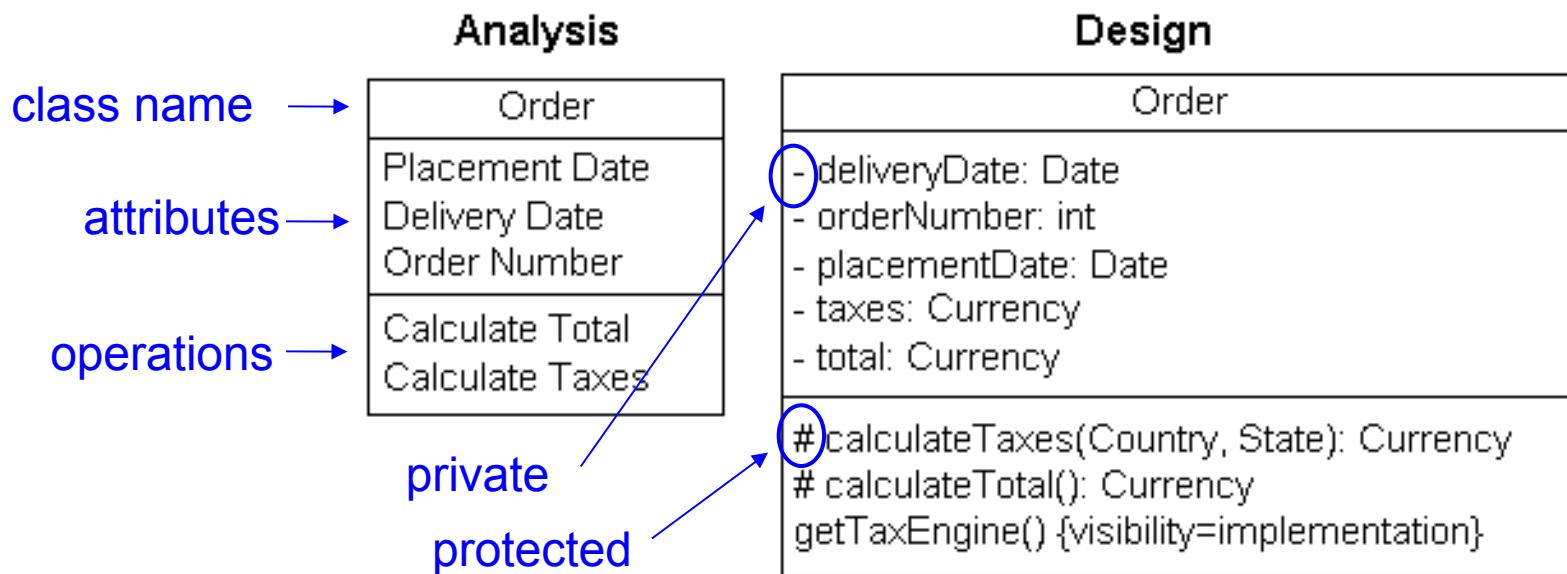
Sources: <https://tallyfy.com/uml-diagram/> <https://www.smartdraw.com/uml-diagram/>
<https://www.lucidchart.com/blog/types-of-UML-diagrams>
<https://creately.com/blog/diagrams/uml-diagram-types-examples/>

Behavior diagrams

- ❖ **Activity diagram** Depicts the control flow: flow of different activities and actions.
- ❖ **State machine diagram** is used to describe the different states of a component within a system.
- ❖ **Use case diagram** Shows the system functionality: the system's high-level requirements.
- ❖ **Communication diagram** shows communication between objects in a form of a system structure.
- ❖ **Interaction overview diagram** is an activity diagram made of different interaction diagrams (timing, sequence, communication).
- ❖ **Sequence diagram** describes the sequence of messages and interactions that happen between actors and objects.
- ❖ **Timing diagrams** is used to represent the relations of objects when the center of attention rests on time.

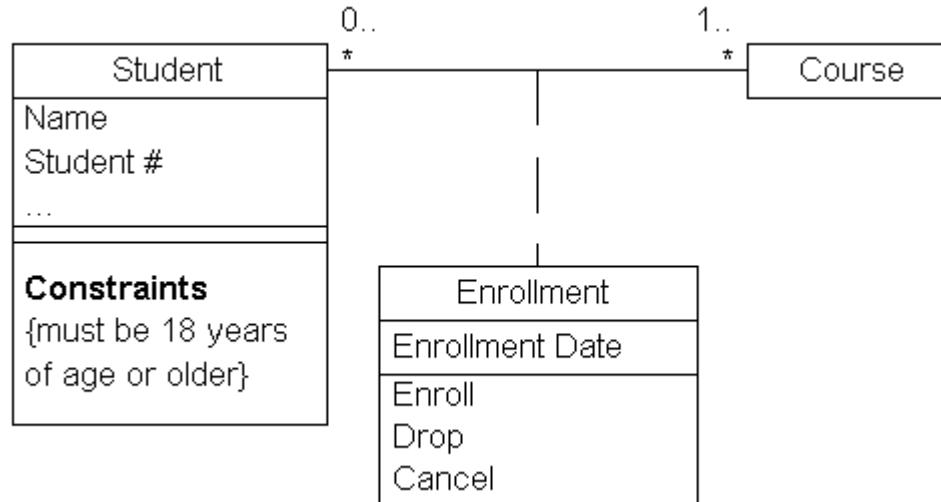
Class diagram

- ❖ Serves for many purposes, from requirements understanding to detailed model of a system to be developed. Can have different styles.



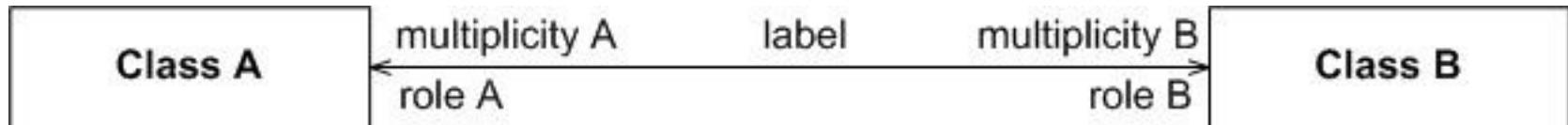
Src.: <http://www.agilemodeling.com/style/classDiagram.htm>,
http://en.wikipedia.org/wiki/Class_diagram

Class diagram: association class



- ❖ association classes are linked with a dotted line.
- ❖ Connections of association classes are not named.
- ❖ The class and dotted connecting line must be positioned centrally to both connected classes.
- ❖ Convenient for data modeling (can replace ER diagram)

Class diagram: connections

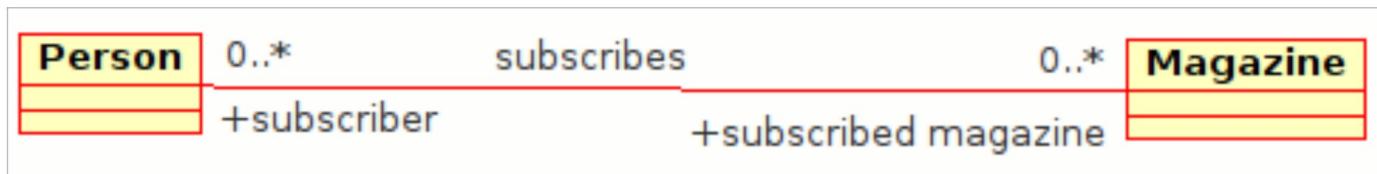


❖ Multiplicity

- ◆ 0..1 zero or one
- ◆ 1 ena
- ◆ 0..* zero or more
- ◆ 1..* one or more
- ◆ n always n ($n > 1$)
- ◆ 0..n zero to n ($n > 1$)
- ◆ 1..n one to n ($n > 1$)

Class diagram: relations (1)

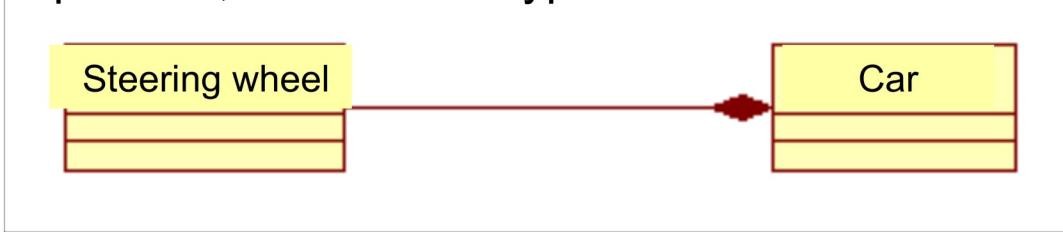
Association



Aggregation, a relation of type “has a”

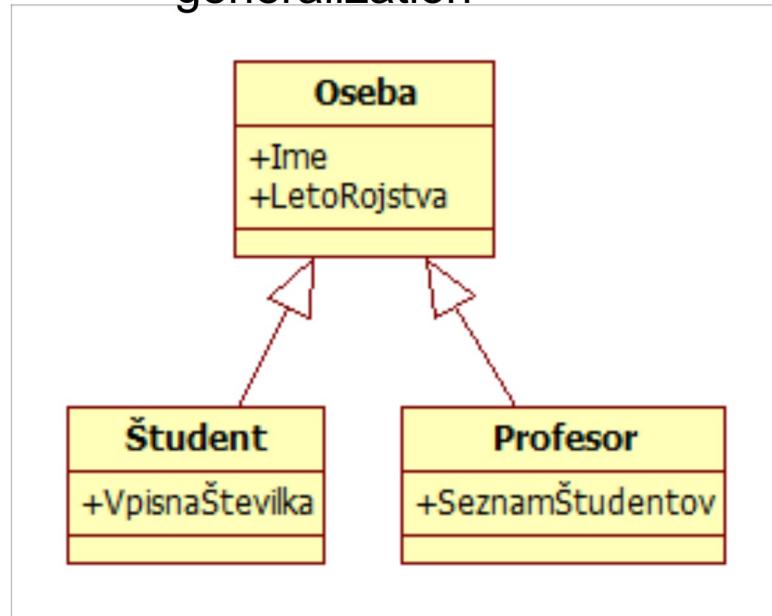


Composition, a relation of type “owns a”

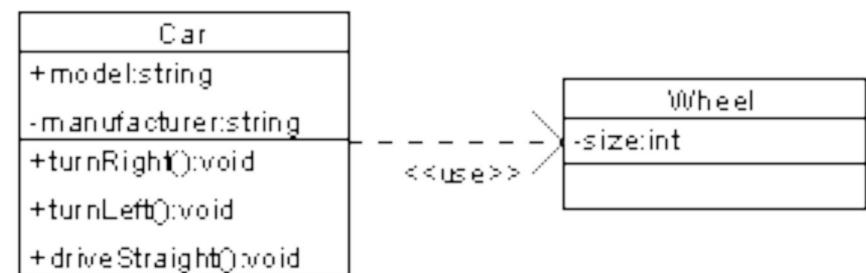


Class diagram: relations (2)

generalization



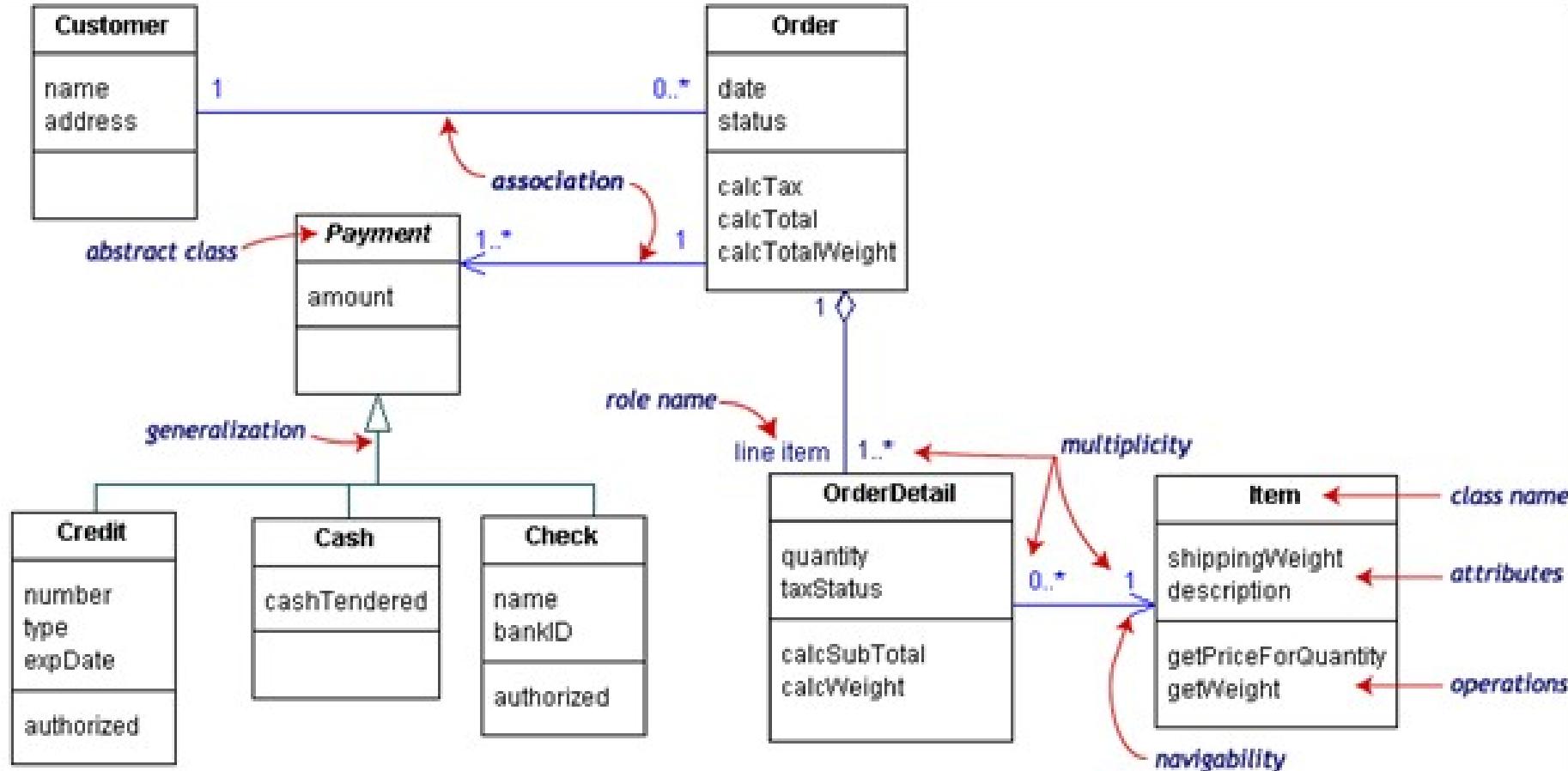
dependency



Class diagram: recommendations

- ❖ Class names and attributes should be unshortened nouns, operations should be named by verbs.
- ❖ Do not model the class skeleton (scaffolding code), i.e., attributes and operations to provide the basic functionality (relations with other classes and get/set functions).
- ❖ Never use two sections – use (one or) three -else it is difficult to distinguish between attributes and operations.
- ❖ Eventual additional sections should be named, e.g., requirements.
- ❖ If the class is not shown completely, label this (e.g., with “...”).
- ❖ Static operations and attributes shall be listed before the others.
- ❖ Exceptions can be provided in operation descriptions, e.g.,
 - ◆ *+ findAllInstances(): Vector {exceptions=NetworkFailure, DatabaseError}*

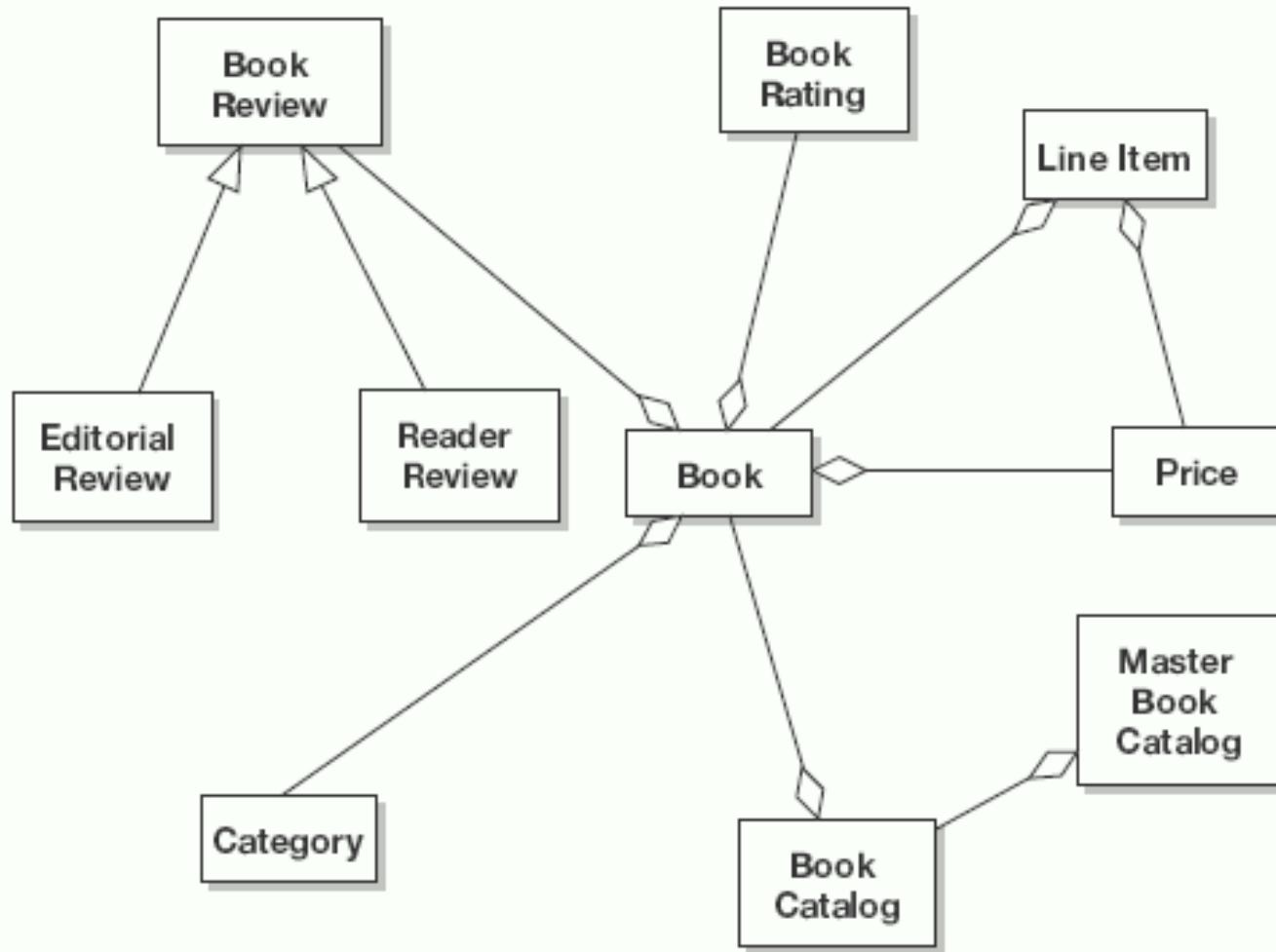
Class diagram: example



class diagram: domain model

- ❖ Domain model is a simple class diagram intended for quick initial description of a system in the phase of analysis.
- ❖ Includes only generalizations and aggregations.
- ❖ Usually does not include attributes and operations (else only the most important ones).

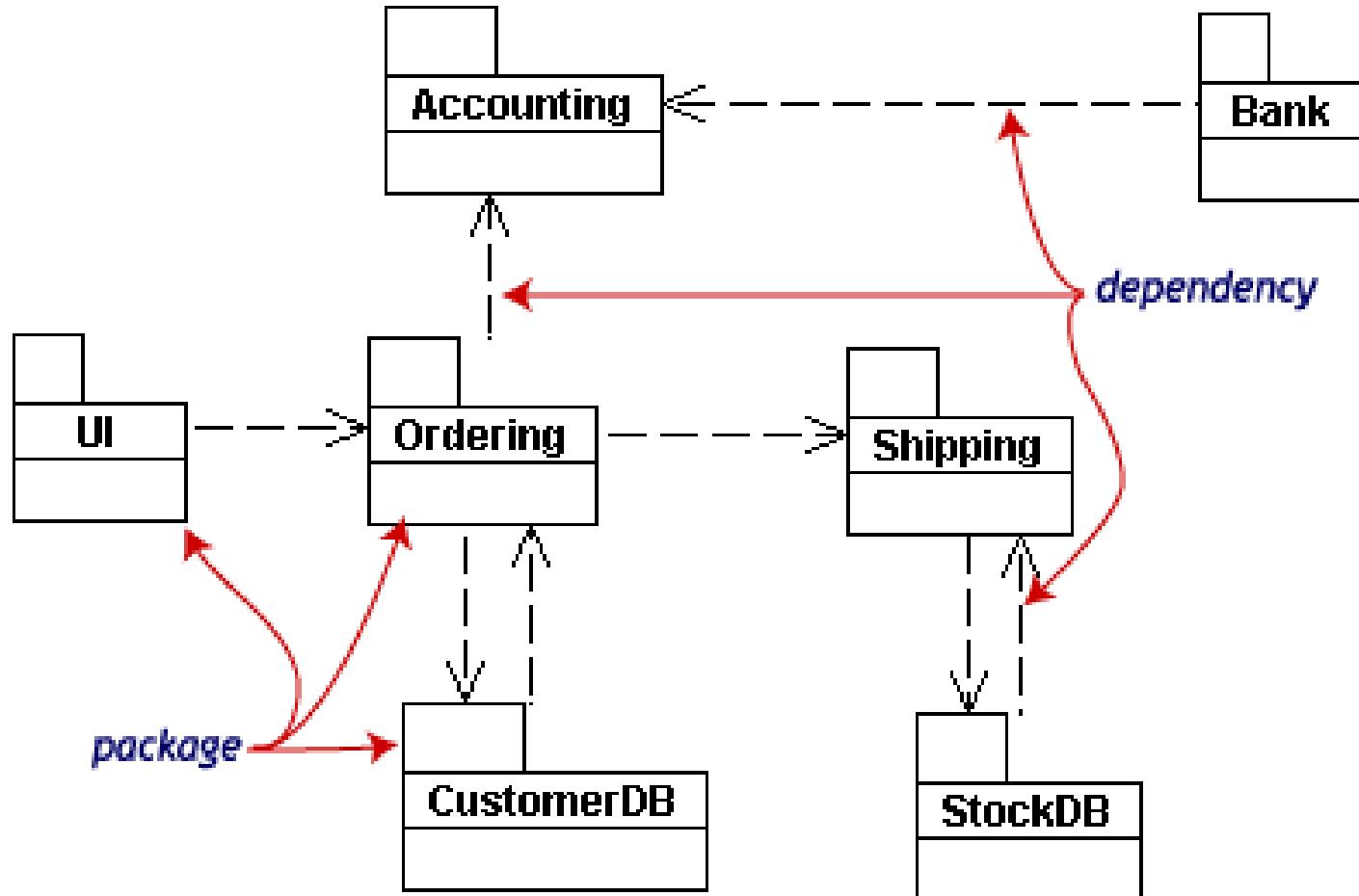
class diagram: domain model



UML package diagram

- ❖ Used to simplify complex class diagrams, where classes are grouped into **packages**.
- ❖ A package is a collection of logically related UML elements (typically classes).
- ❖ Packages are connected by their dependencies. A package is dependent of another package if a change in one may require change in the other.

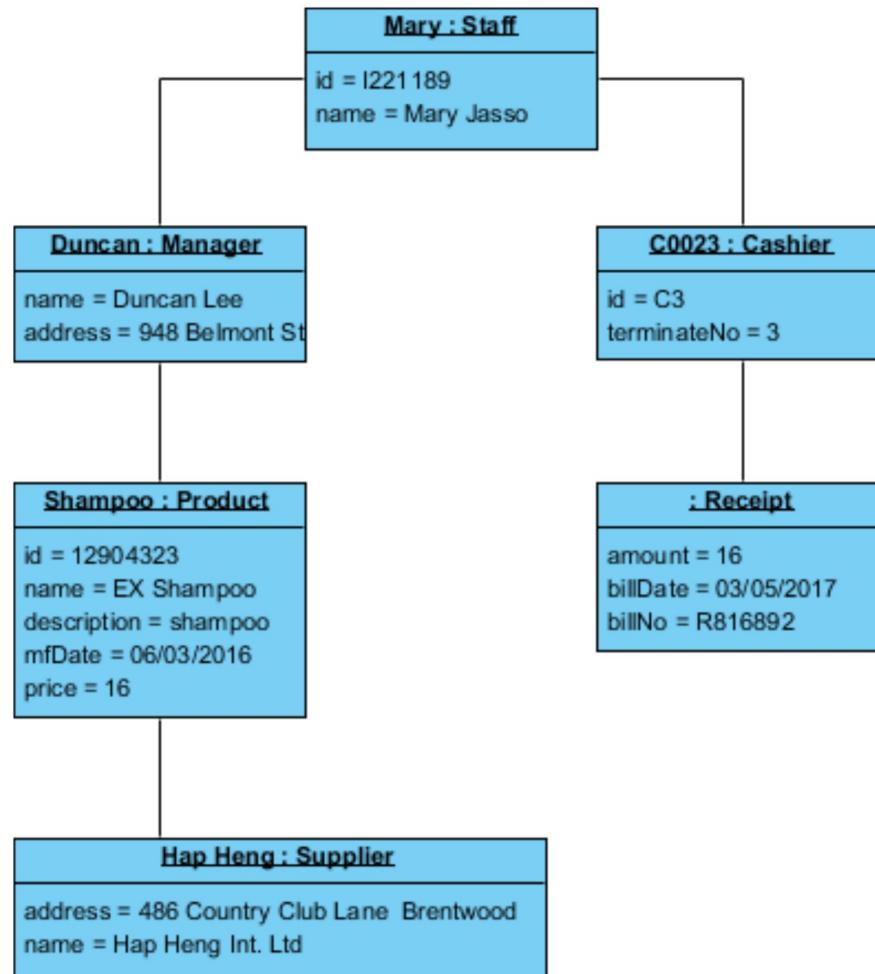
UML package diagram: example



Object diagram

- ❖ Object diagrams are suitable for depicting class usage.
- ❖ They are showing “real world” objects and their relations.
- ❖ Needed when class diagrams are too abstract.
- ❖ Object diagrams shows an example of class usage in one time during SW operation.

Object diagram: example



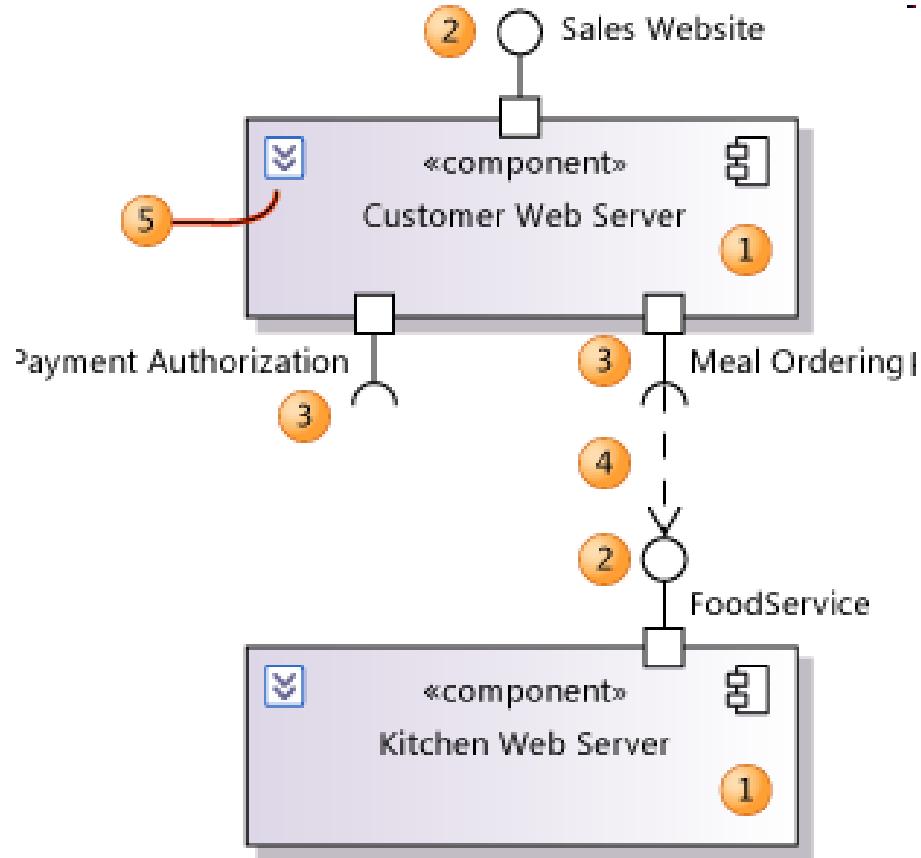
Src.:<https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-object-diagram/>

Component diagram

- ❖ Component diagram shows components and their connections.
- ❖ A component is a modular unit with clear interfaces and corresponding ports.
- ❖ Some components offer ports while others may need them for their operation.
- ❖ Internal properties of components are hidden and inaccessible.

Component diagram: elements

1. A component
2. Interface port offered by a component
3. Interface port that a component needs for its operation
4. Connection between the components.



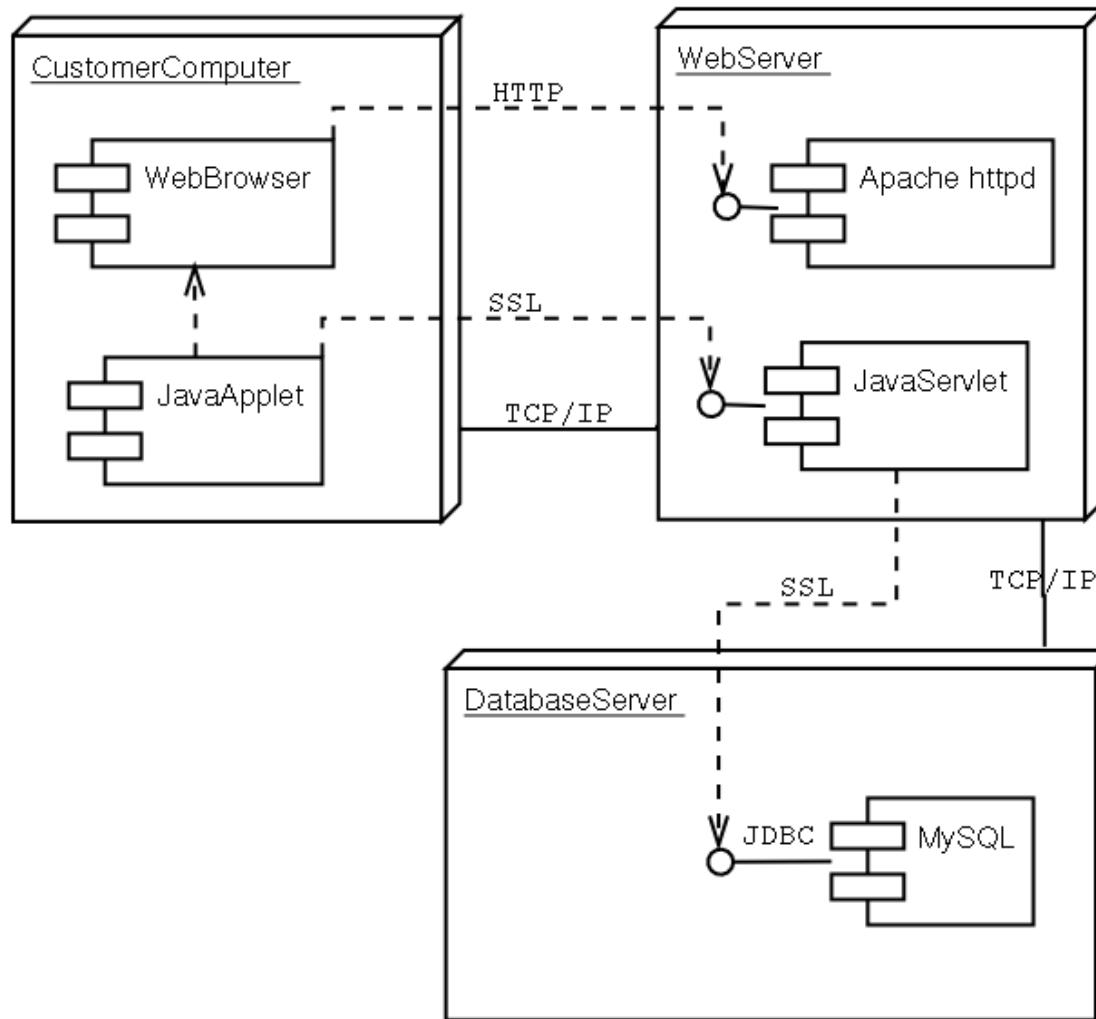
Additional literature:

http://www.disi.unige.it/person/ReggioG/ISII04WWW/COMPONENTDIAGRAM_MOD.ppt

Deployment diagram

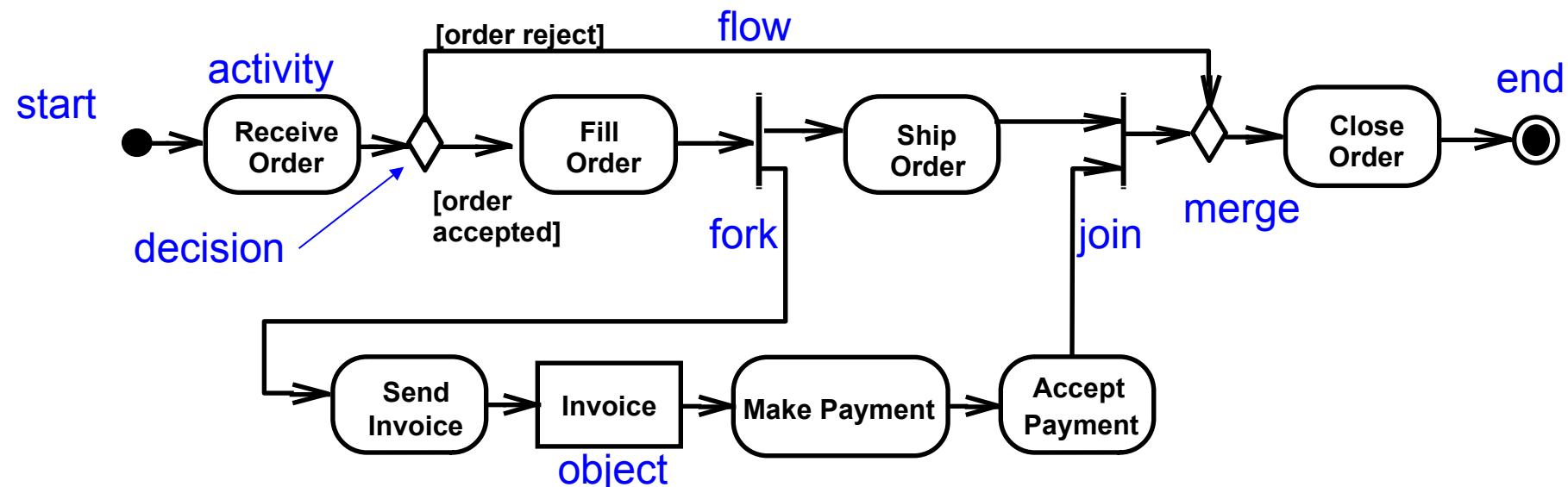
- ❖ Deployment diagram shows physical relations between system HW and SW.
- ❖ HW elements:
 - ◆ Computers (servers, clients)
 - ◆ Embedded devices
 - ◆ Other devices (sensors, perrifery)
- ❖ Deployment diagram shows HW used by the SW.

Deployment diagram: example



Activity diagram

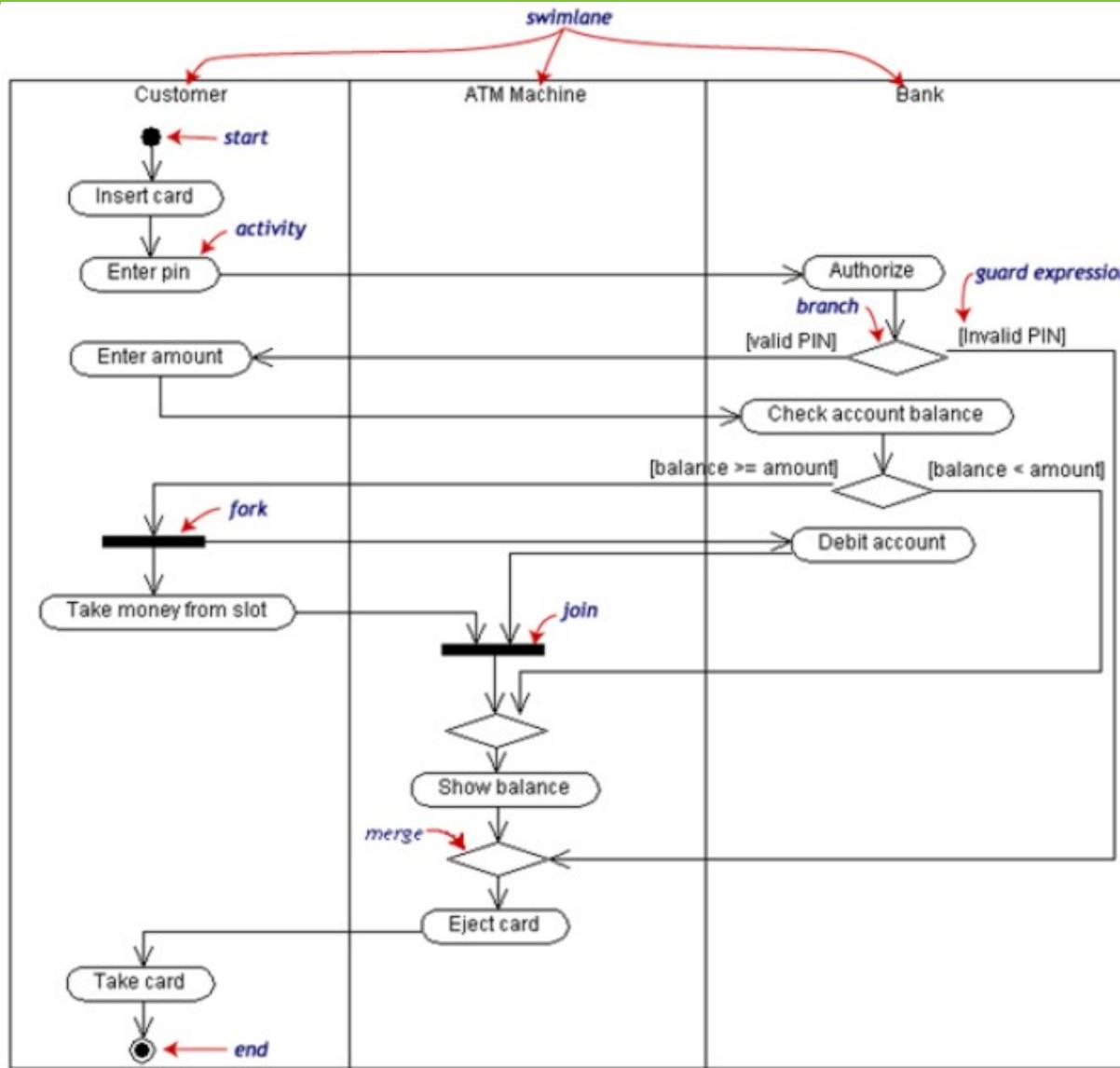
- ❖ Activity diagram is an UML variant of flowchart.
- ❖ Models a control flow at SW execution.



Additional literature:

<http://www.disi.unige.it/person/ReggioG/ISII04WWW/ActivityDiagrams.ppt>

Activity diagram: example 2

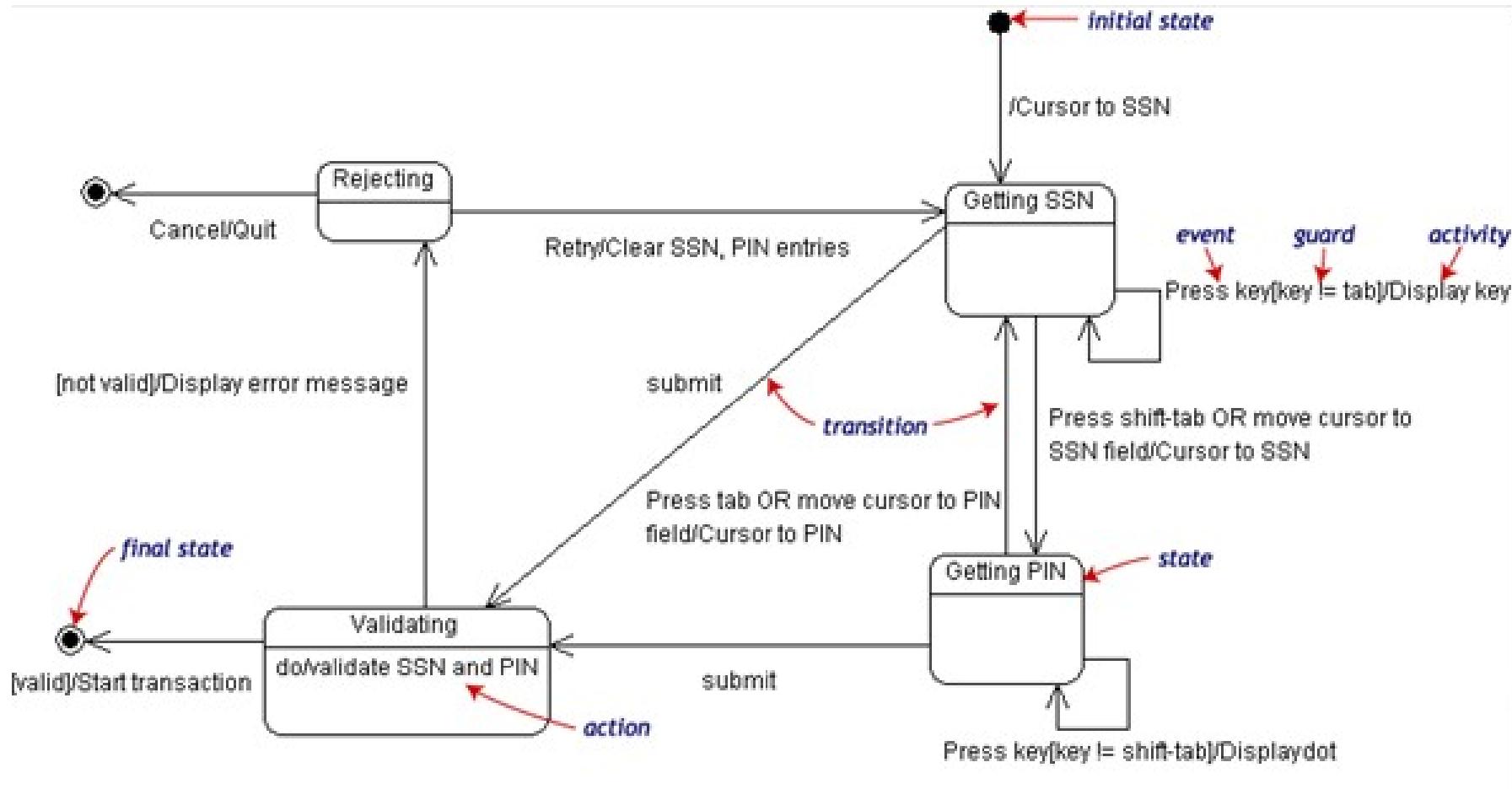


Src.: <http://edn.embarcadero.com/article/31863>

State machine diagram

- ❖ Some objects act differently dependent to a state in which they are in certain moment.
- ❖ An object state depends on previous activities and changes due to (external) events.
- ❖ A state can be associated with a characteristic activity, that executes at transition to this state.
- ❖ The state machine diagram depicts possible object states and transitions between the states, events that influence state changes, and activities related to states and transitions.

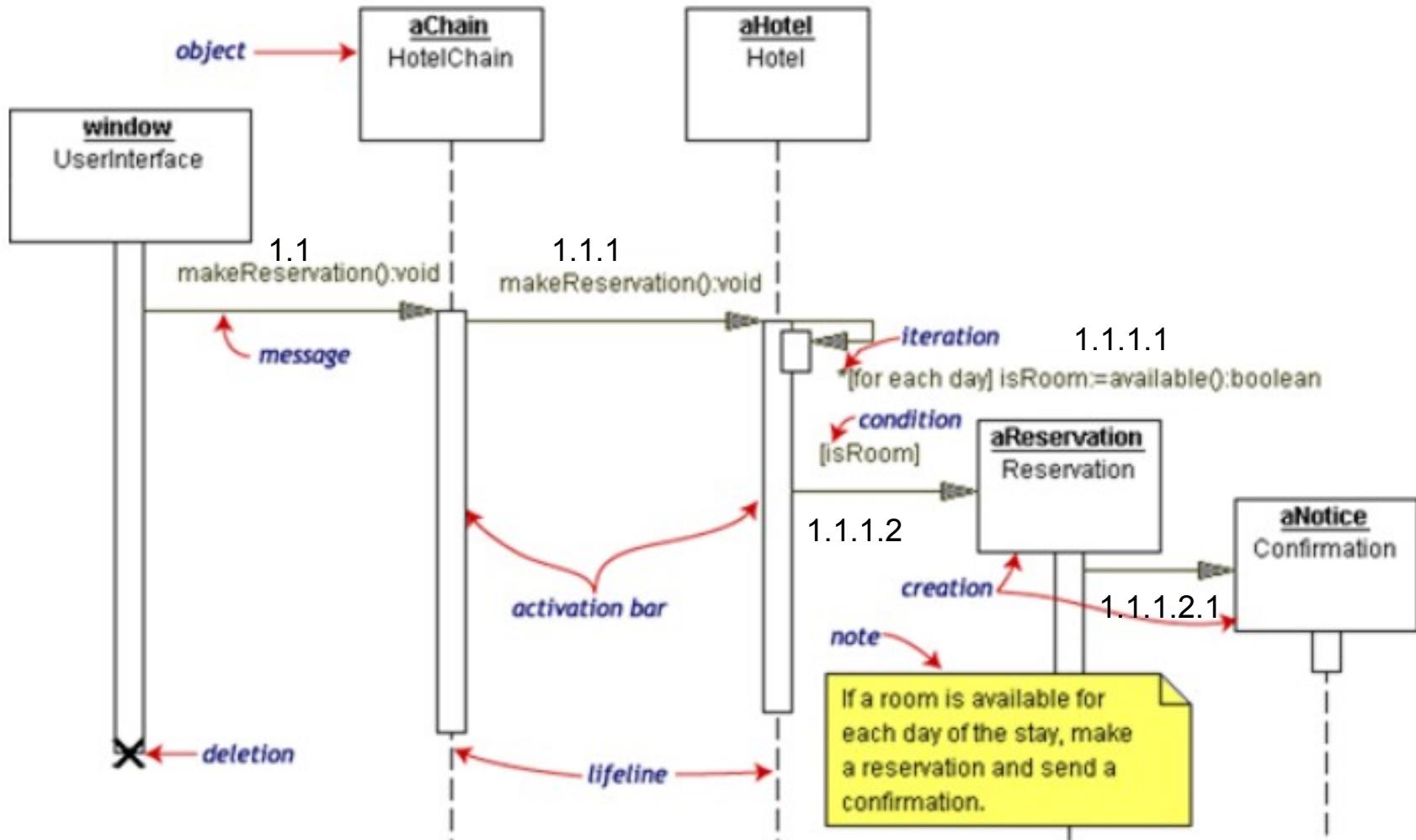
State machine diagram: example



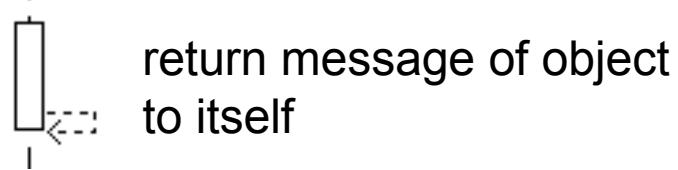
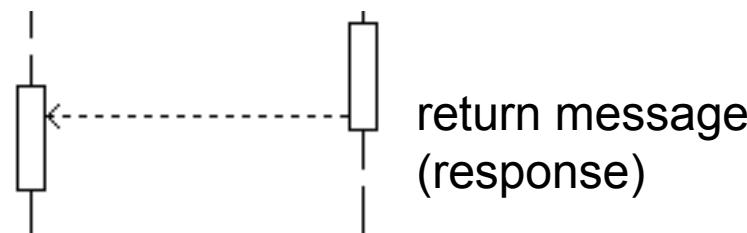
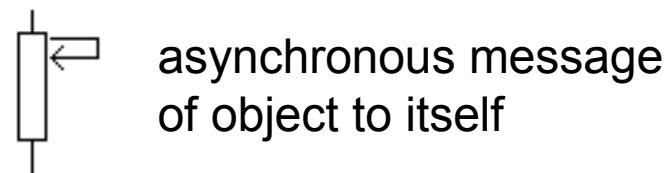
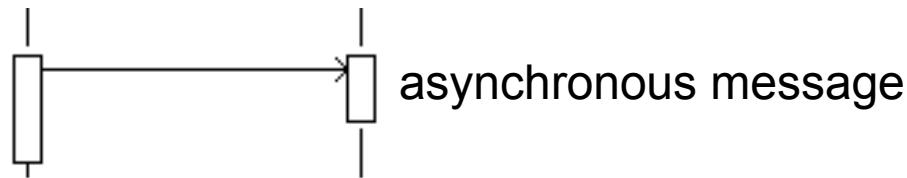
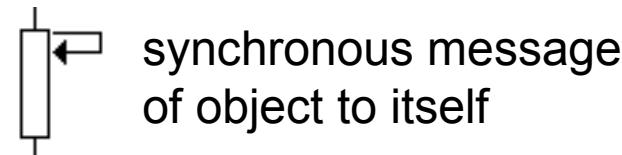
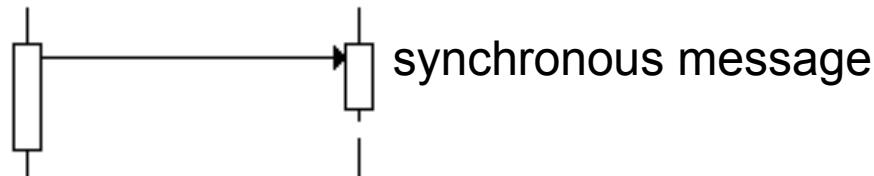
Sequence diagram

- ❖ The sequence diagrams show communication between objects as a sequence of messages.
- ❖ Each object has its own lane that corresponds to time of object existence and labels object activities.
- ❖ Suitable for conceptual and logical models, at different levels of abstraction.
- ❖ Additional literature:
http://www.tracemodeler.com/articles/a_quick_introduction_to_uml_sequence_diagrams/

Sequence diagram: example



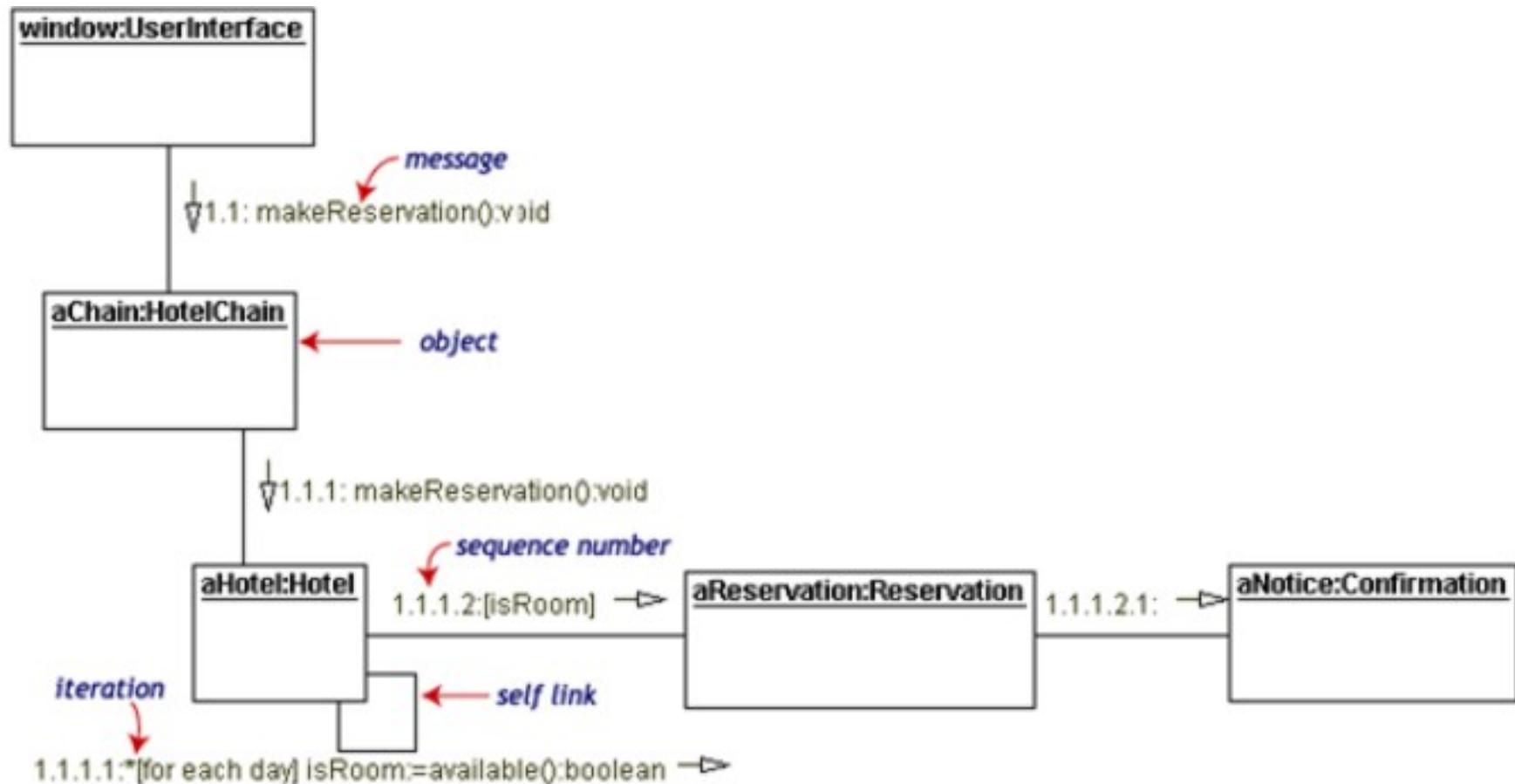
Sequence diagram: messages



Communication diagram

- ❖ Communication diagram - UML 1.x name it collaboration diagram.
- ❖ It is equivalent to sequential diagram, provide the same information.
- ❖ More focused to connections between objects than temporal sequence of messages.

Communication diagram example

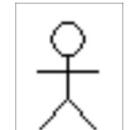


Use case diagram

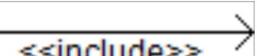
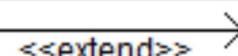
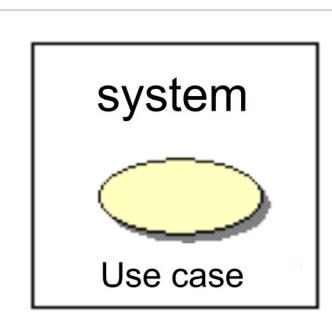
- ❖ The use case diagram provides an external view on the system and depicts user requirements.
- ❖ Shows interaction between users and the system.
- ❖ Defines the system boundary (what is included in the system and what is not)
- ❖ It does not correlate with system structure or implementation!

Use case diagram: symbols

- ❖ Actor is a part of environment, executes the use case and has some advantage of it. It can be an individual person, other system, other device, etc.
- ❖ System boundary is shown with a rectangle
- ❖ A use case is a typical group of scenarios that are needed to reach a user goal.
- ❖ A scenario (activity) is a sequence of steps describing interaction of an actor and the system.
- ❖ Connections (communication, extension inclusion, generalization)

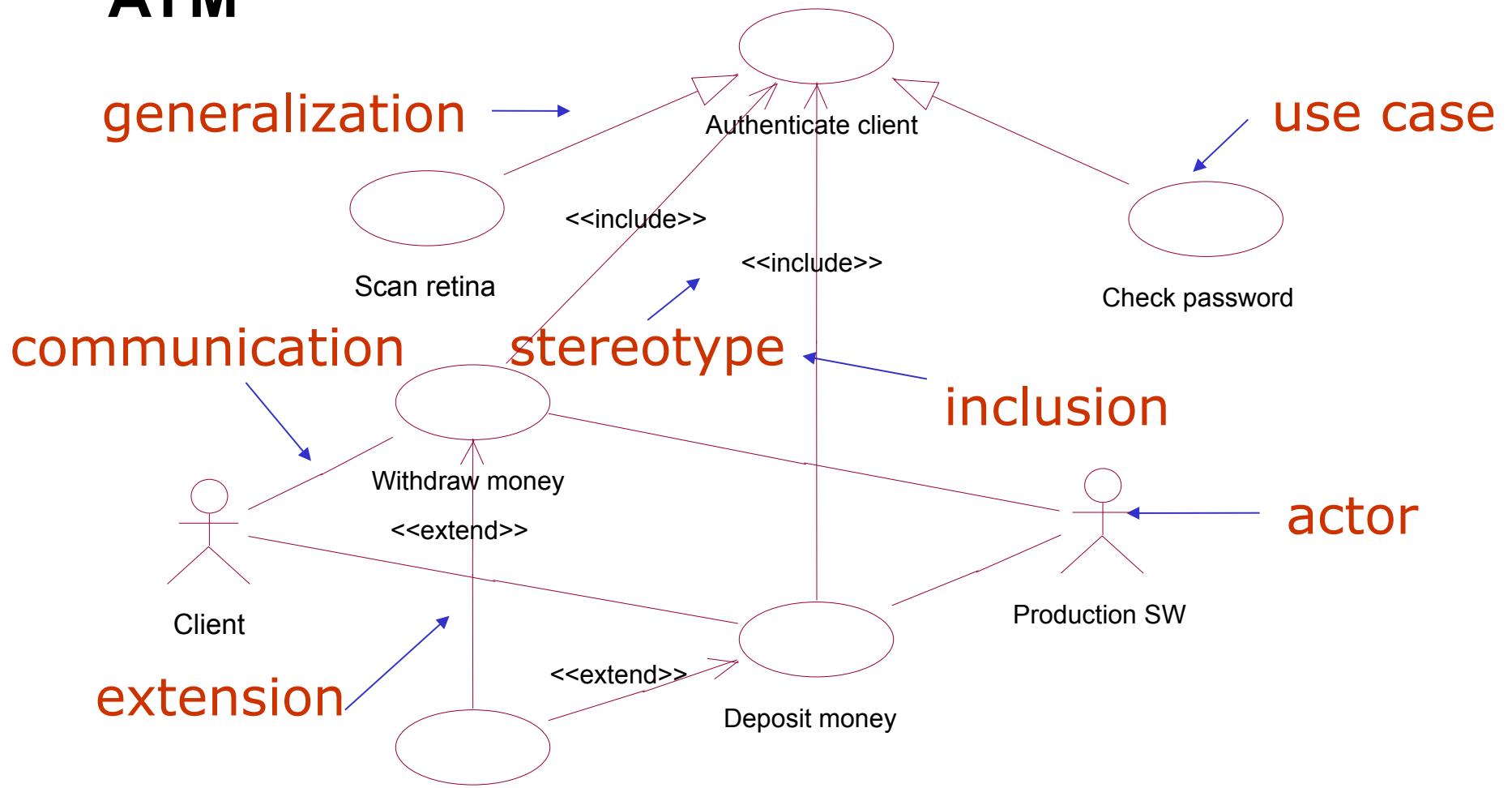


actor

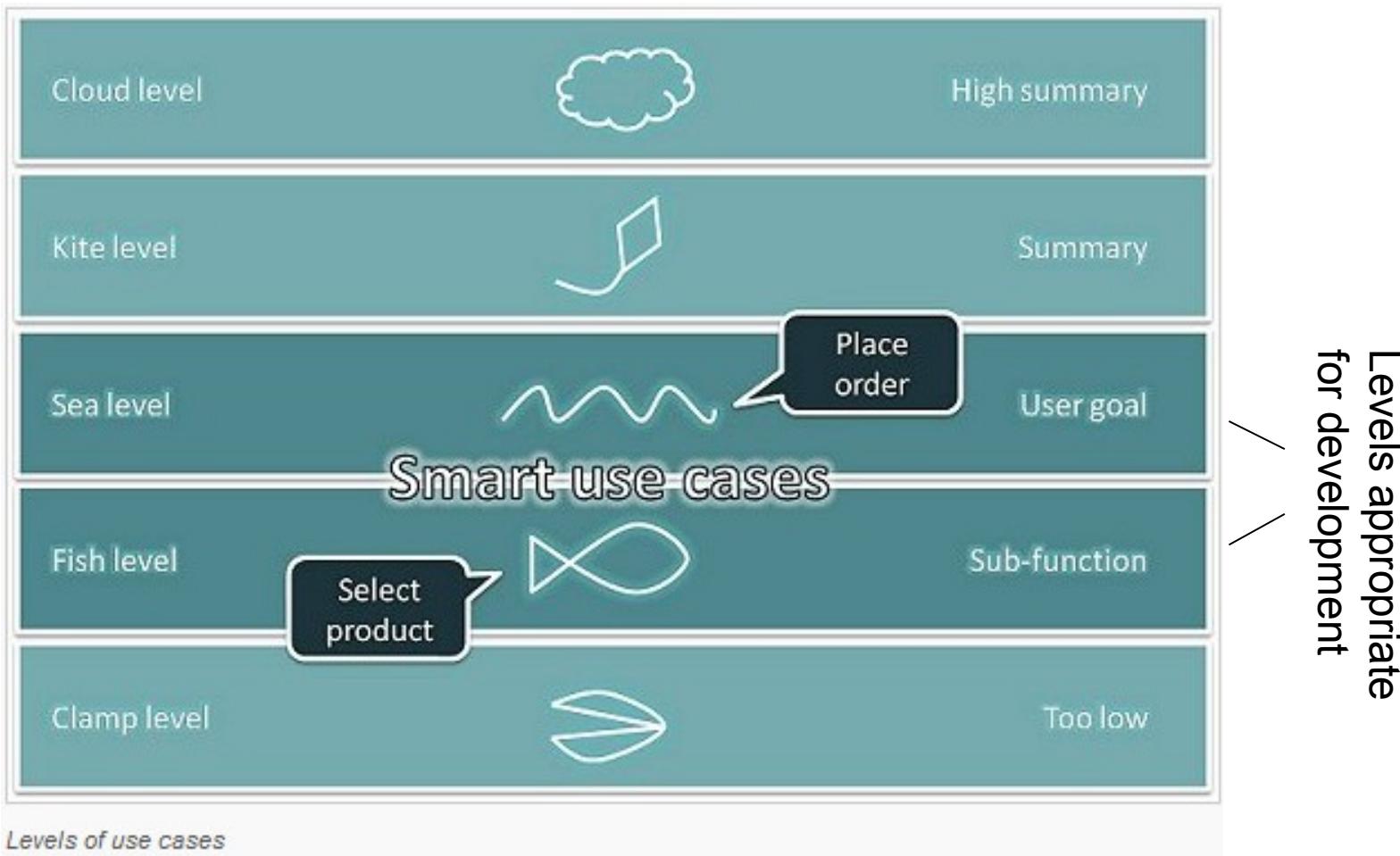


Use case diagram: example

ATM



Use case diagram: levels



Src: <https://www.youtube.com/watch?v=FNkuGER1gB4>

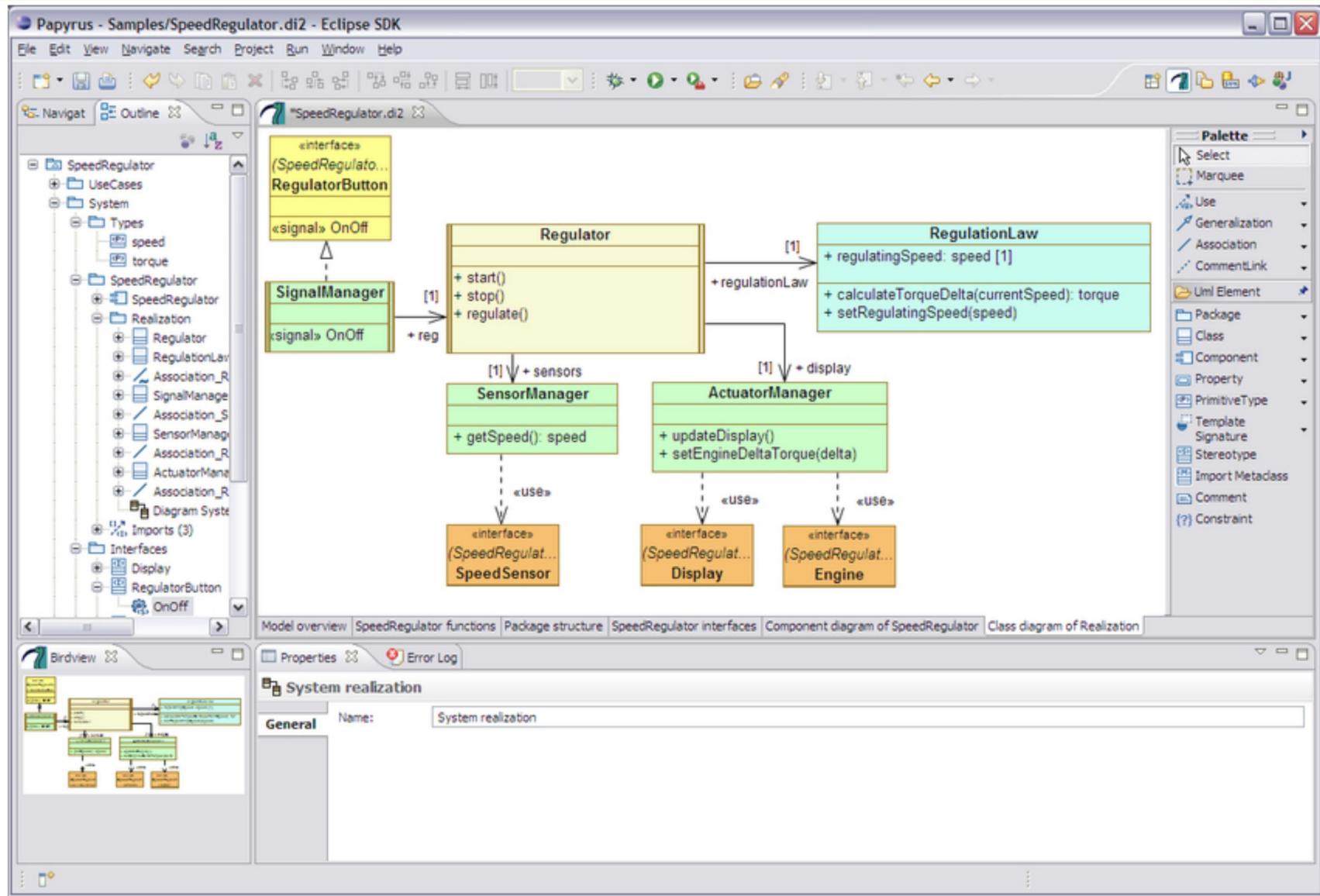
Use case diagram: recommendations

- ❖ Use cases shall depict activities needed to reach the user goals and not activities related to system operation.
- ❖ The main actor shall be located top left.
- ❖ Actor does not use other actors.
- ❖ Use case does not use other use case, but it can include it or extend it.
- ❖ A diagram includes at least one actor and at least one use-case.

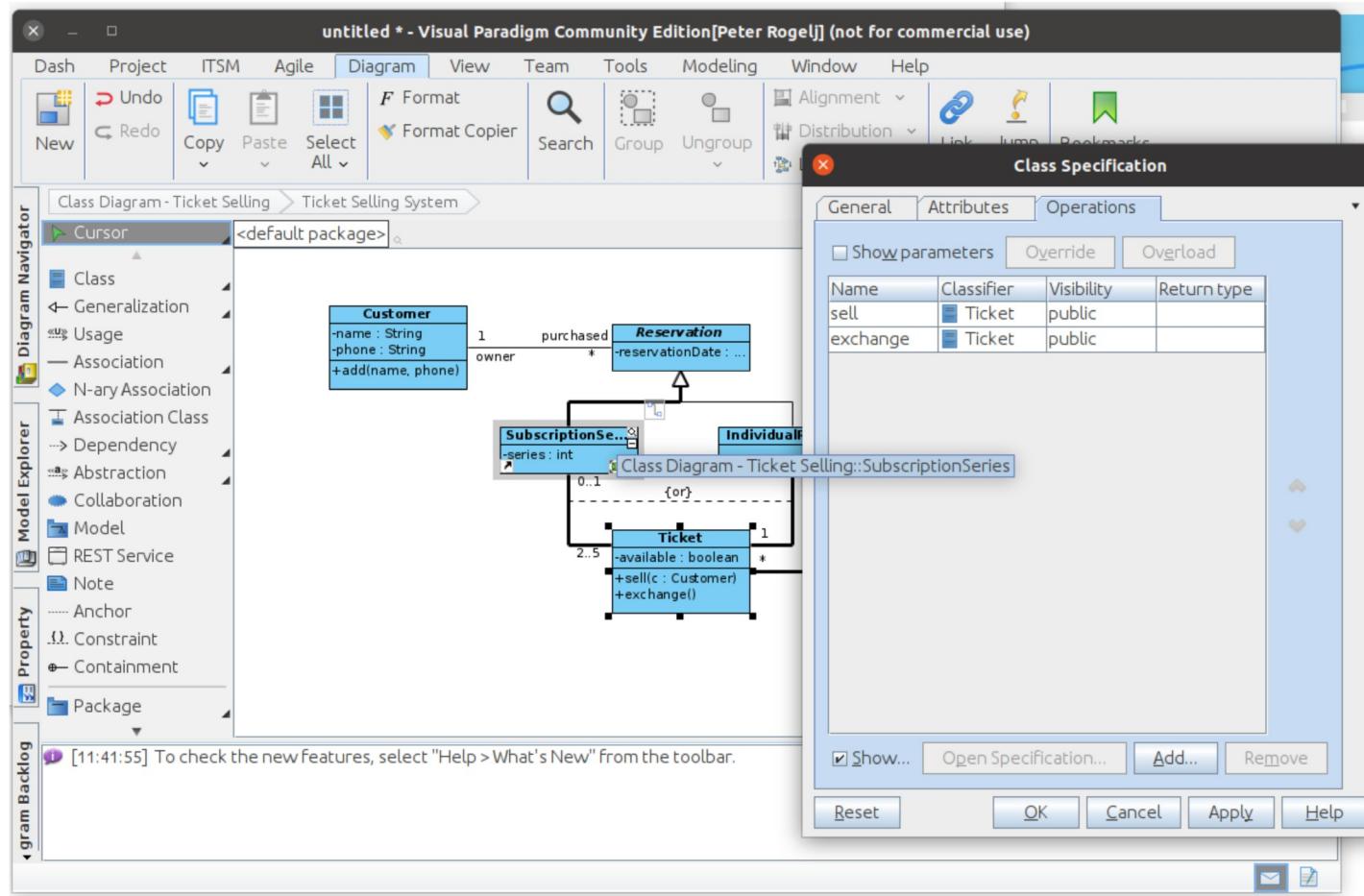
UML diagrams

- ❖ Use diagrams that help solving the problem – development of a system.
- ❖ Do not go into too much details.
- ❖ Standard notation makes diagrams more readable. Some freedom remains at the syntax.

UML tools: Eclipse + Papyrus



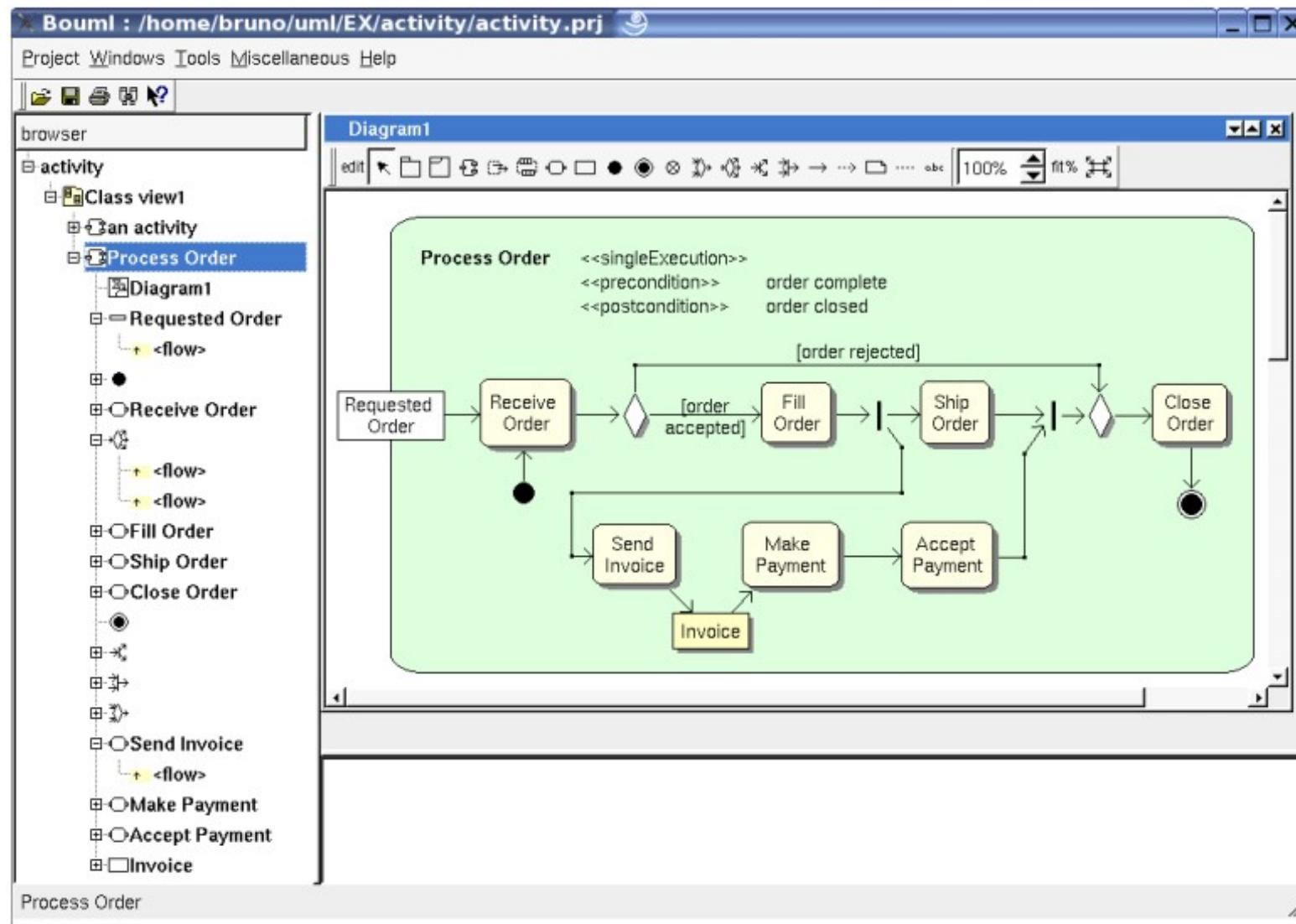
UML tools: Visual paradigm



Community edition: <https://www.visual-paradigm.com/download/community.jsp>

Online: <https://online.visual-paradigm.com/>

UML tools: BOUML



UML tools

There are several other tools enabling modeling based on UML:

- Enterprise Architect (Sparx):

<https://sparxsystems.com/products/ea/index.html>

- Altova Umodel:

<https://www.altova.com/umodel>

- Modelio: <https://www.modelio.org/>

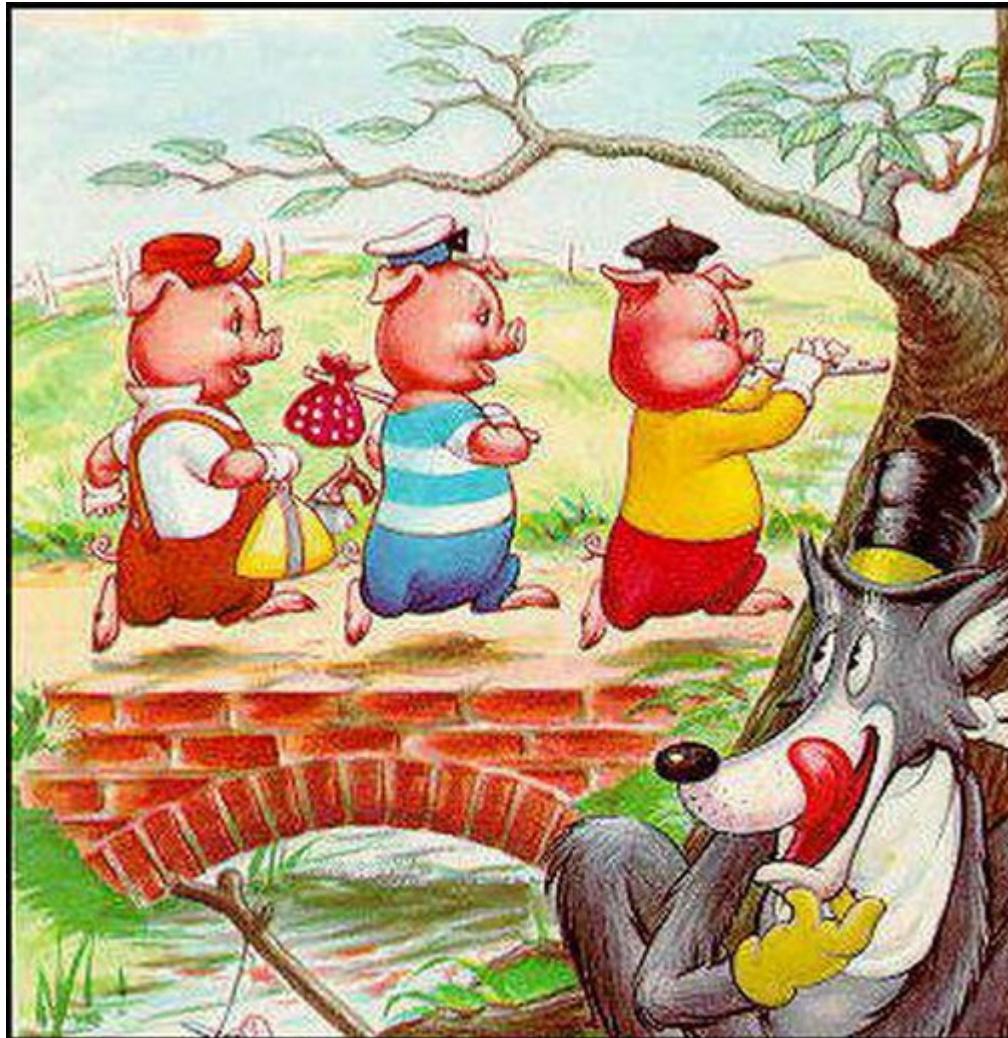
...

Check the prices.

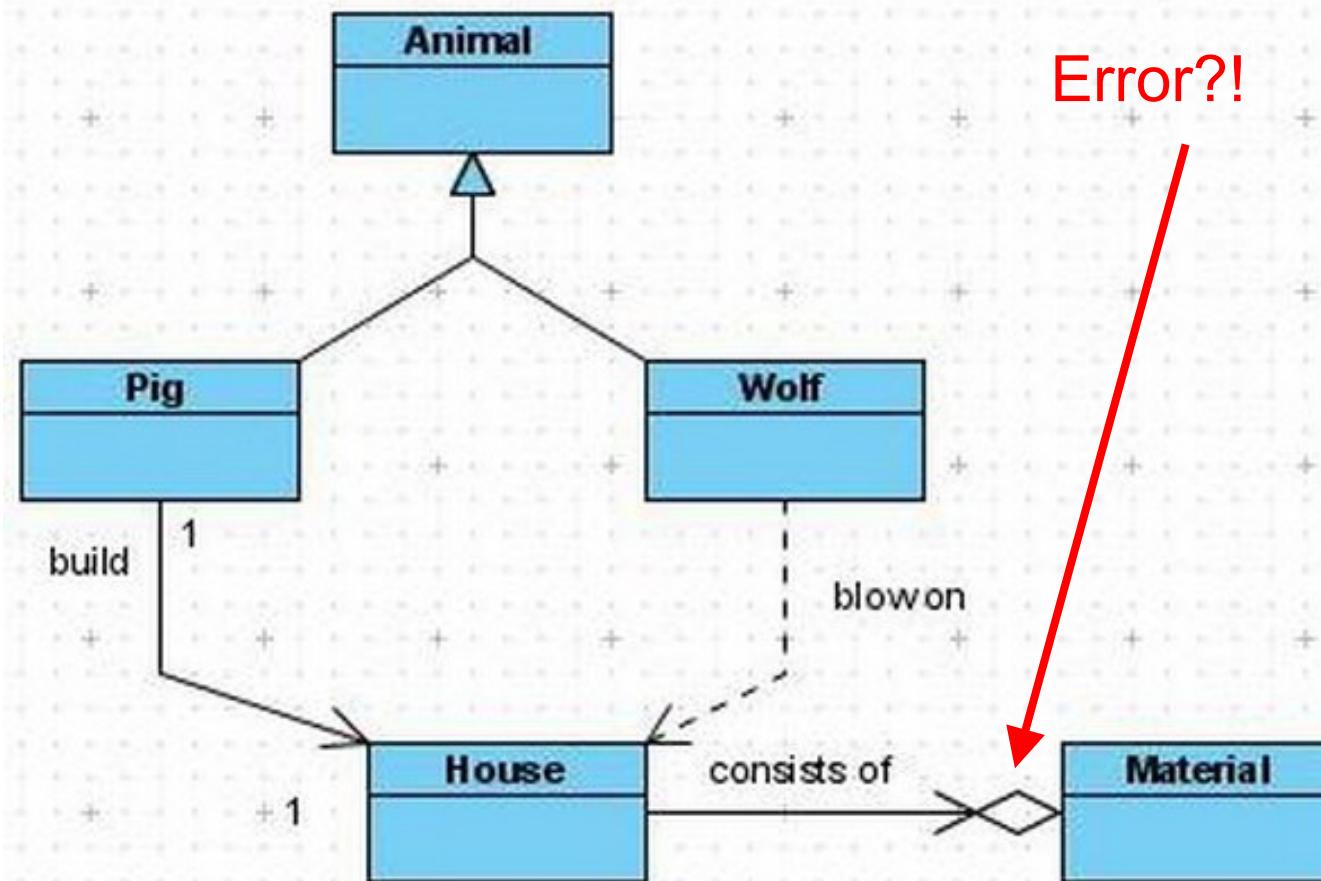
Diagramming techniques conclusion

- ❖ Modeling is important but the extent of modeling depends on actual needs (SW type, size, organization...)
- ❖ UML is a recognized standard, but for certain specific models some other techniques may be more popular:
 - ◆ Database modeling: ER
 - ◆ Data flow modeling: DFD

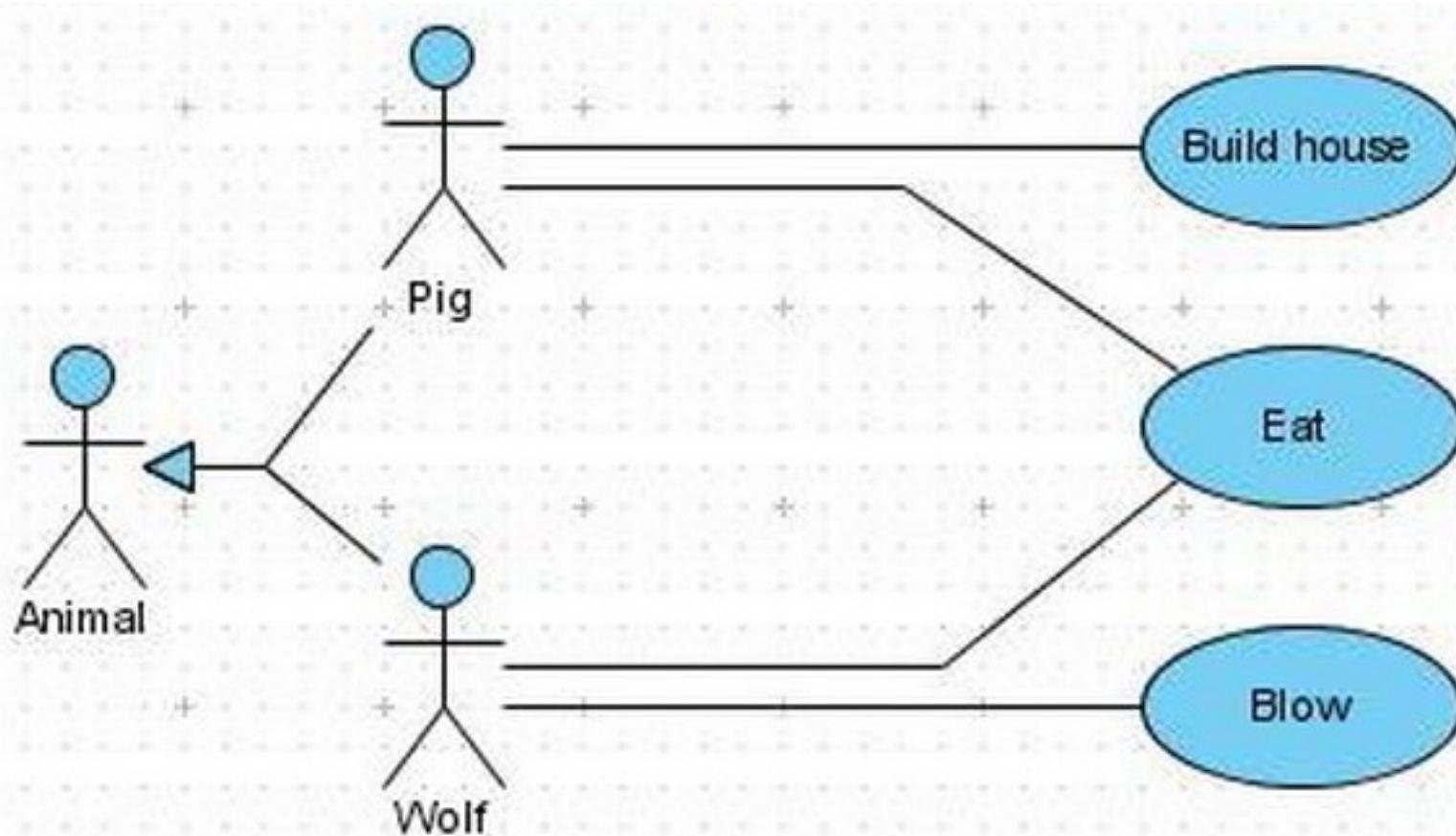
Examples (1A)



Examples (1B)

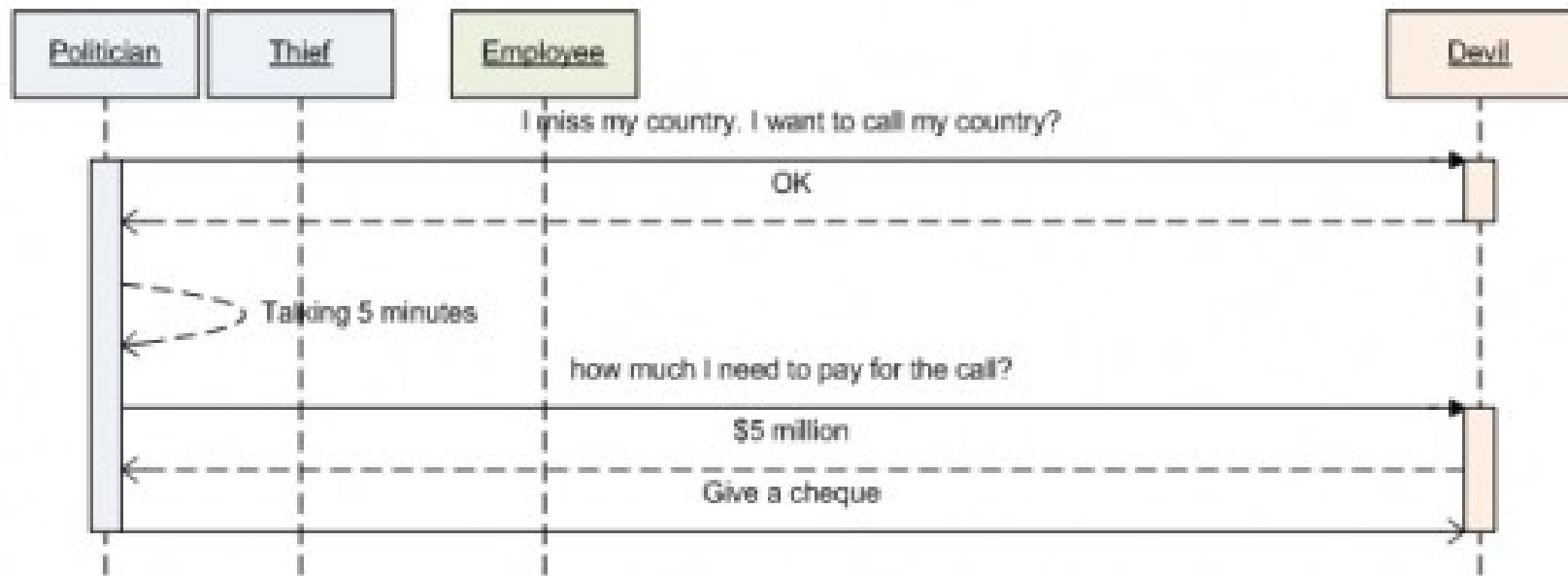


Examples (1C)



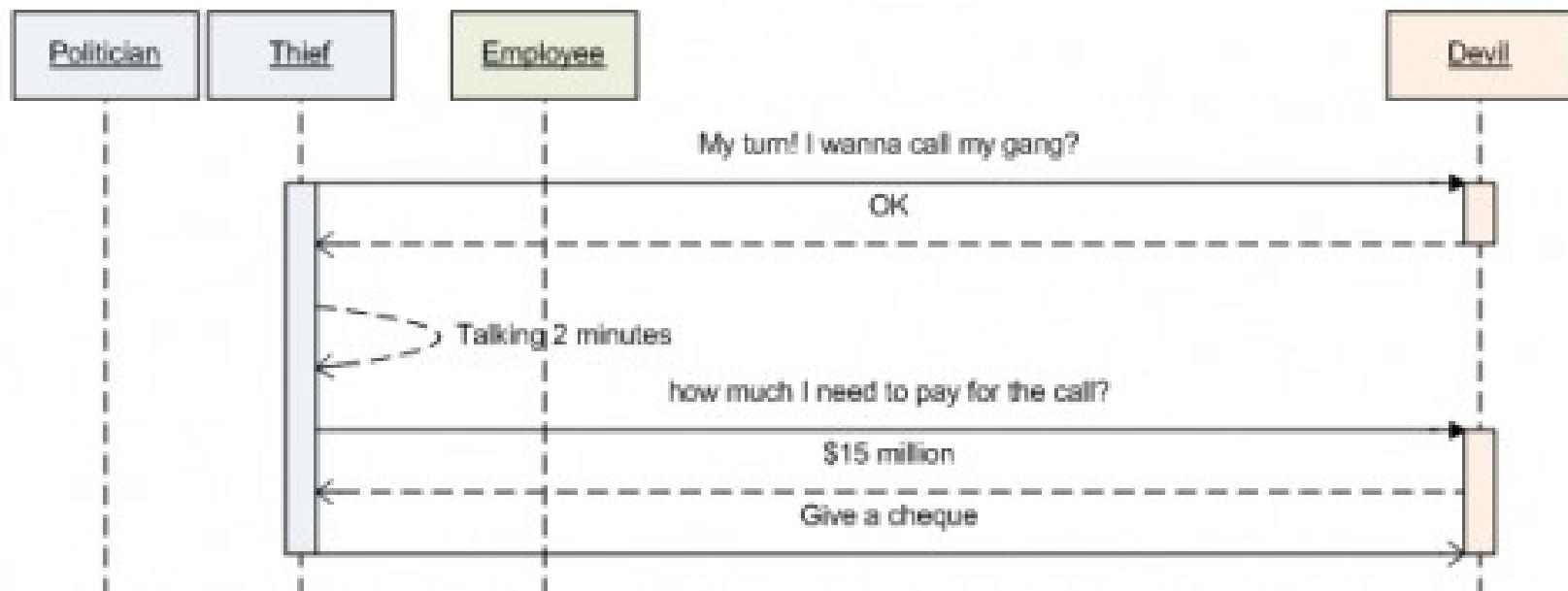
Examples (2A)

Underworld Anecdote



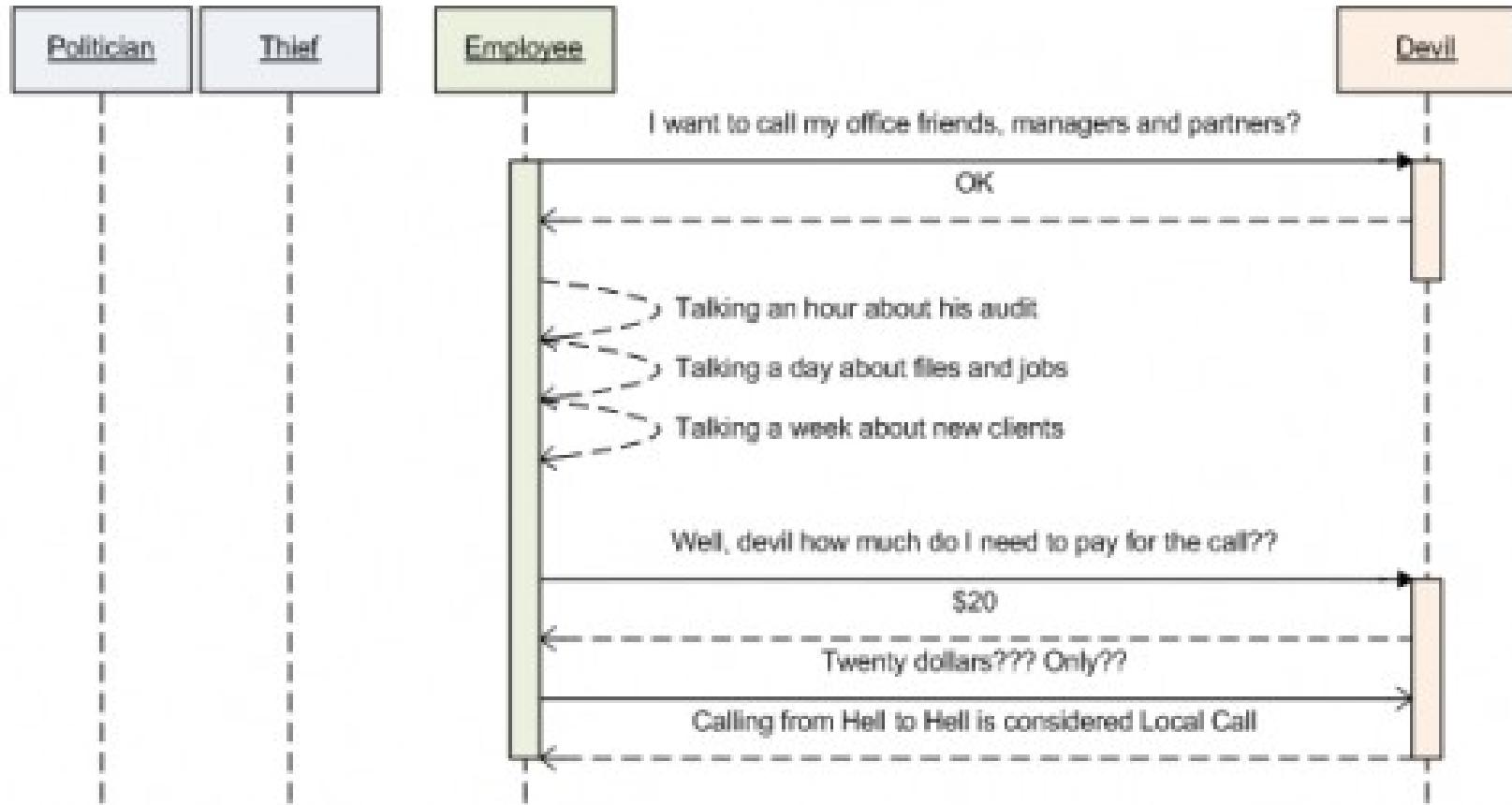
Examples (2B)

Underworld Anecdote



Examples (2C)

Underworld Anecdote



Software engineering

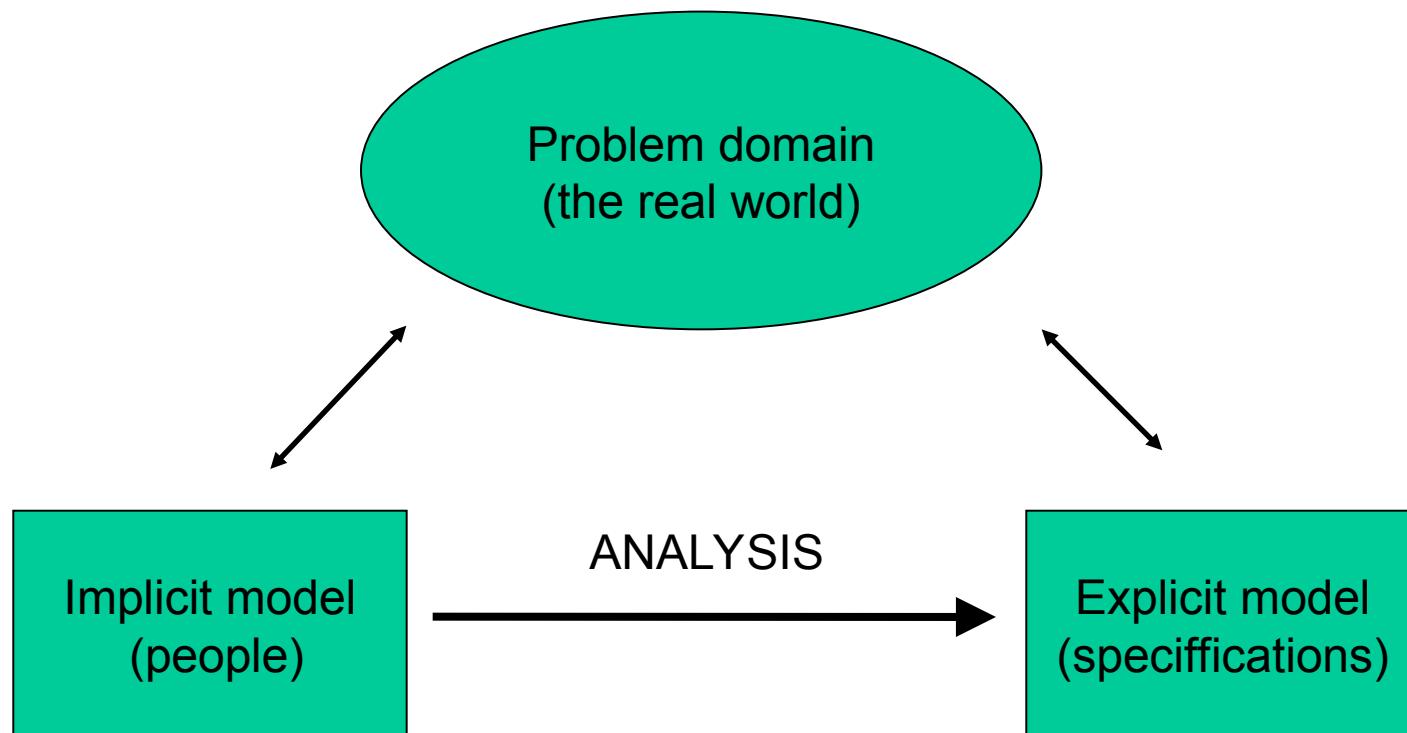
Requirements analysis

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Requirements analysis and definition

- ❖ Also called System Analysis – performed by system analyst.
- ❖ The goal is to define clear explicit requirements.
- ❖ The input information is a real problem description, which is typically poorly defined, unclear and often contradictory.
- ❖ The challenge is to define such a system that
 - ◆ meet user needs and
 - ◆ meet user expectations.

The goal of analysis



Problems to be solved

- ❖ Implicit models are not verbalised.
- ❖ Implicit models are changing with time.
- ❖ Clients and a system analyst do not speak the same “language”.
- ❖ All implicit knowledge cannot be expressed formally.

The expected result

- ❖ The result of analysis is a System Requirements Specification (SRS) – a complete description of system behavior:
 - ◆ Functional requirements: use cases for all interactions of users with the system.
 - ◆ Other – nonfunctional requirements: requirements that limit system design or implementation, e.g., performance, quality standards...
- ❖ Incorrect, incomplete or inconsistent specification is often the source of project failures and disputes.

Nonfunctional requirements

- ❖ Nonfunctional requirements can be divided:
 - ◆ Operational requirements
 - Security, safety, usability...
 - ◆ Evolutionary requirements
 - Testing ability, maintenance ability, extensibility, upgradability...
- ❖ For examples of nonfunctional requirements see:
http://en.wikipedia.org/wiki/Non-functional_requirement

SRS usage

- ❖ Software Requirements Specification (SRS) is needed in order to:
 - ◆ To familiarize with user needs and limitations.
 - ◆ To reach similar understanding of the problem and solution for both, the analyst and the users (customers).
 - ◆ To define the basis for system design.
 - ◆ To define the basis for system validation.

The procedure

1. Getting familiar with the problem and collecting of requirements.
2. Analysis, modeling, coordination.
3. Recension (validation of requirements)
4. Requirements management.

Familiarizing with the problem and Collecting of requirements,

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Requirements collection

- ❖ May look simple:
 - ◆ Ask the customer and users:
 - What is the purpose of the system?
 - What are the goals?
 - How shall the system fulfill the business requirements?
 - How will the system be daily used?
- ❖ It is not as simple as it seems!
 - ◆ How to define the system scope?
 - ◆ How to correctly understand requirements?
 - ◆ What to do if the requirements change by time?

The system scope

- ❖ The scope may not be clear:
 - Who will directly use the system?
 - Which other systems will be connected with this system?
 - Which functionality shall be included (and which not)?
- ❖ The customer may provide technical details
 - ❖ that may not be required and may even not be technically correct
 - ❖ may that may make understanding of the system more difficult.

Requirements understanding

- ❖ Customers/users may not be sure what they need:
 - ◆ “Make it such that it will help us.”
 - ◆ “Try to improve the problem solution.”
- ❖ They may not know limitations of their HW, SW or processes well.
 - ◆ They do not have a full understanding of the problem domain.
 - ◆ They cannot define requirements to system analyst.
 - ◆ They leave out information that seems obvious for them.
 - ◆ They list opposing requirements.
 - ◆ They list unclear requirements and requirements that cannot be verified.
- ❖ System analyst is often not an expert in the problem domain.

Requirements changes

- ❖ Requirements do change by time
- ❖ Reasons are:
 - ◆ Organizational structure changes (new products, new departments...)
 - ◆ Changes in operational extent (increased number of daily transactions, higher number of user, higher communication throughput...).
 - ◆ External changes
 - Changing of law (taxes...)
 - Changes of international standards (MPEG...)
 - ◆ Customers get additional ideas when they get familiar with the capabilities of a new system.

Information gathering steps

- ❖ Study of literature
- ❖ Expert advice
- ❖ Questionnaires
- ❖ Derivation from existing systems
- ❖ Synthesis from environment
- ❖ Using of prototypes

Study of literature

- ◆ Books from the problem domain
- ◆ Process descriptions
- ◆ Process improvement studies
- ◆ Development documentation
- ◆ Error and system logs, bug tracking.
- ◆ Proposals

Expert advice

- ❖ Listen to experts from the problem domain
- ❖ Experts may not be only those linked to the customer.
- ❖ Expert advantages
 - ◆ They know the problem domain in detail
 - ◆ Can help defining the scope.
 - ◆ Can direct system analyst to alternative solutions.

Questionnaires

- ❖ System analyst asks customers and/or users
 - ◆ what they expect from the new system,
 - ◆ expecting that they will not be too biased or limited by their limitations of understanding.
 - ◆ In addition to questionnaires interviews and brainstorming sessions are possible.
 - ◆ Communication may be direct (oral) or indirect (written).

Derivation from existing systems

- ❖ The analysis is started based on an existing system.
 - ◆ the system to be replaced,
 - ◆ a similar system somewhere else (other organization),
 - ◆ a system described in a literature.

Synthesis from environment

- ❖ SW can be successful only if the properties of the system environment are considered.
- ❖ This is called process analysis or normative analysis.

Using prototypes

- ❖ In use when requirements cannot be clearly defined elsewhere.
- ❖ Prototype is a working model of a part of a system or the whole system.
- ❖ Prototypes differ from the final system also due to non-compliance with nonfunctional requirements.

Atomic requirements

- ❖ Requirements are atomic if:
 - ◆ Precisely define the need without being divided into more requirements.
 - ◆ Are measurable.
 - ◆ Are testable.
 - ◆ Are traceable

Analysis, modeling, coordination

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Analysis, modeling, coordination.

- ❖ When the requirements are gathered, the following steps must be made:
 - ◆ Analysis
 - Requirements analysis in a sense of consistency, completeness and reality.
 - ◆ Modeling
 - Building an explicit model of requirements in a form of natural language, diagrams and formal notations (mathematic equations, pseudo code...)
 - ◆ Coordination
 - The proposed solutions must be coordinated with stakeholders regarding the scope, contradicting requirements, priorities and risks.
- ❖ Usually the steps repeat in multiple iterations.

Analysis

- ❖ Categorization of requirements
 - ◆ Grouping request into related subsets.
- ❖ Finding relations between requirements.
- ❖ Testing requirements consistency
- ❖ Identifying missing data
- ❖ Checking requirements ambiguities
- ❖ Defining priorities based on user needs.
 - ◆ Priorities can serve for defining the incremental or phased development.

Questions at req. analysis

- ❖ Is every requirement consistent with the goals?
- ❖ Are all requirements at the same level of abstraction?
 - Too many technical details can limit further development,
 - Missing technical details can leave requirements unclear.
- ❖ Are all requirements needed to reach the goals?
- ❖ Is every requirement clear and unambiguous?
- ❖ Is the source of each requirement clear (who, why)?
- ❖ Are there contradicting requirements?
- ❖ Is each of the requirement technically feasible in the given environment?
- ❖ Will it be possible to test each of the requirement when the SW is developed?

Modeling

- ❖ Requirements can be presented with different kinds of models:
 - ◆ Natural languages
 - Easy to understand.
 - Risks for ambiguities and unclarity.
 - ◆ Mid-formal notations
 - More precise than natural languages.
 - Simple to understand
 - Do not require strictly defined notations.
 - ◆ Formal notations
 - E.g., mathematical expressions.
 - Exact, unambiguous.

Modeling and models

- ❖ Requirements specification models must offer understanding of system from three perspectives:
 - ◆ Data – data model
 - What form of data will be used and how are different pieces of data related.
 - ◆ Processes – process model
 - Which processes form a system and what is their function.
 - ◆ Execution – control model
 - The sequence of executing process activities, decisions and merges, forks and joins of control flow.

Analysis modeling principles

- ❖ There are many different approaches to analysis and modeling, including:
 - ◆ Top-down approach
 - **Struktural analysis**
 - ...
 - ◆ Object oriented approach
 - **Using UML**
 - ...
 - ◆ Other: data driven approach
 - ...

Requirements modeling

❖ Structural analysis

- ◆ List of events
- ◆ Contextual diagram
- ◆ Data flow diagrams (DFD) and specifications
- ◆ Entity relational diagrams (ERD)
- ◆ Activity diagrams in a form of flowchart
- ◆ ELH

❖ Using UML (OOA)

- ◆ Use cases (diagram + description)
- ◆ Domain diagram (class diagram)
- ◆ Sequence diagrams, communication diagrams
- ◆ Activity diagrams, state diagrams

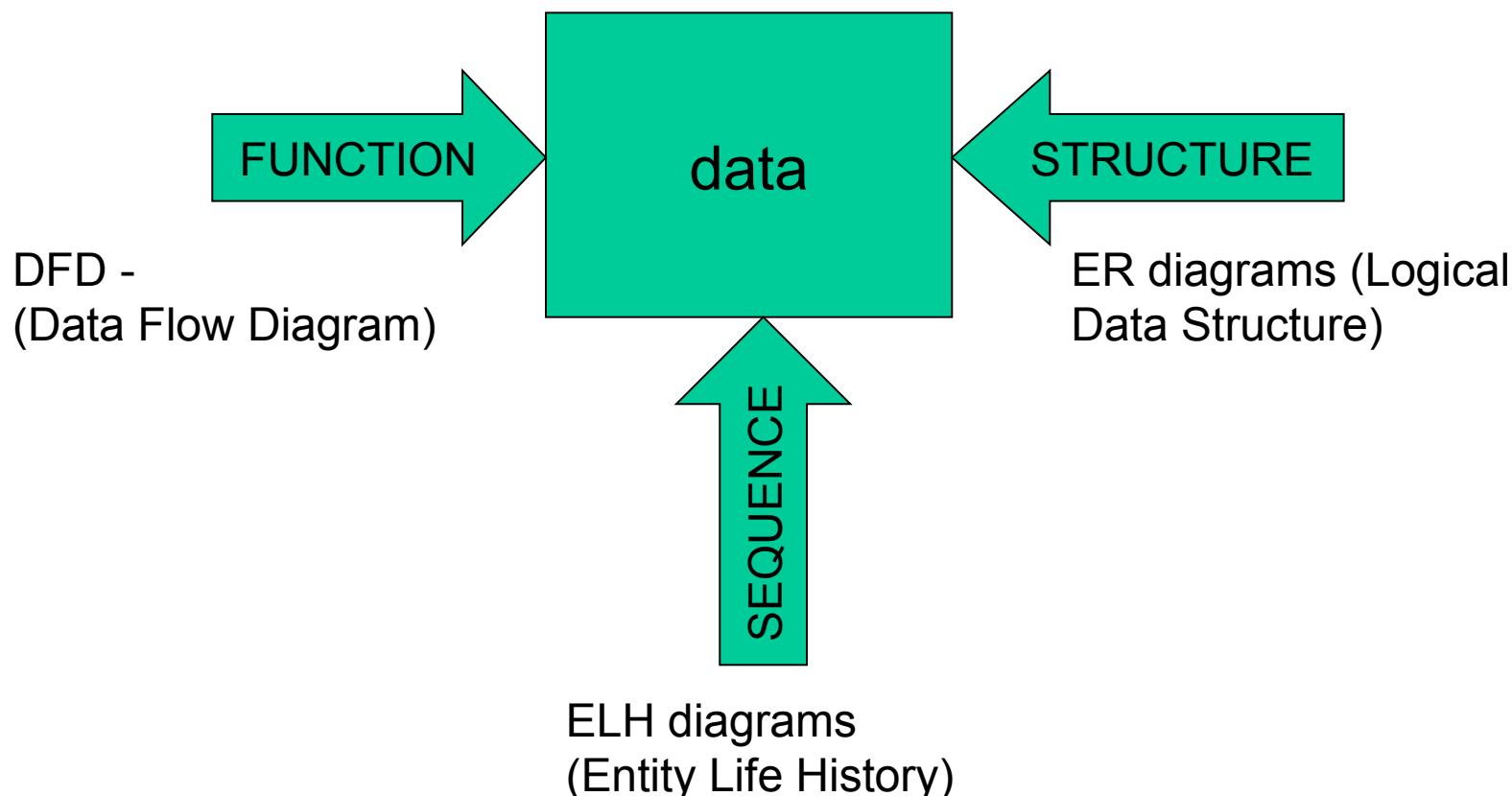
Structural analysis

- ❖ Structural analysis
 - ◆ Discussed at classes of
Systems 3 – Information systems (SIS3-IS)

Example: SSADM

Remember (SIS3-IS):

SSADM (Structured systems analysis and design method) defines three views on systems data:



Event list

- ❖ The list of external events that happen in external environment and have influence on the system.
- ❖ Example:
 - ◆ Student registers for examination.
 - ◆ Teacher publishes lecture summary.
- ❖ Each event is described in a scenario.

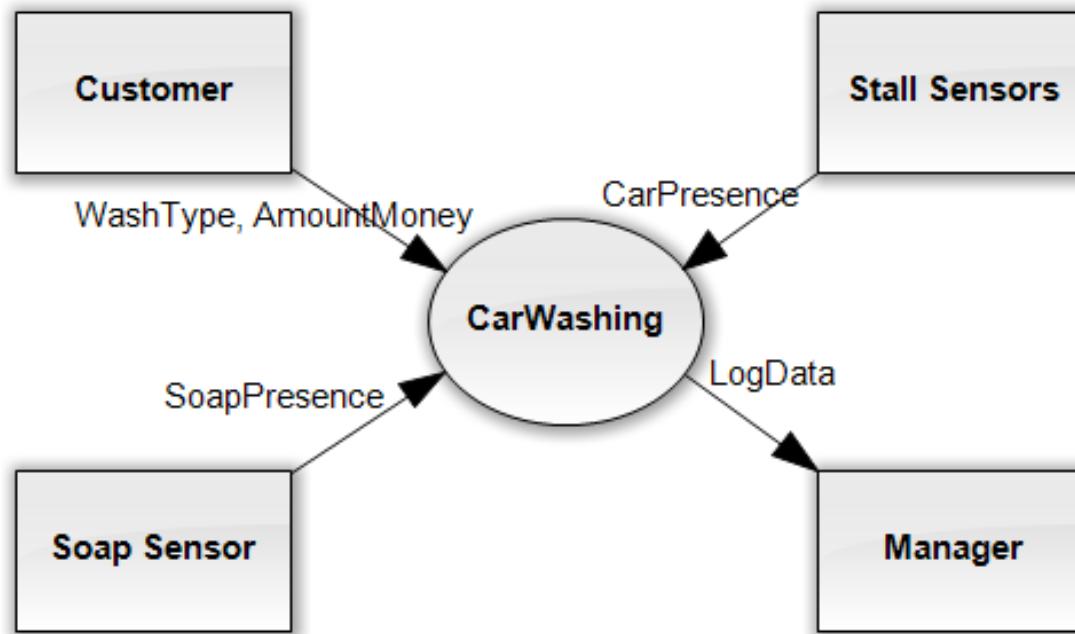
Scenario example (car wash)

Michael drives to the kiosk outside the carwash and attempts to insert five dollars. The kiosk accepts the money and displays the message “You have paid enough for a complete wash. Insert another dollar for a deluxe wash.” Michael presses the button for an express wash. The kiosk refunds one dollar. The carwash queries its stall sensors. The stall sensors report that no car is detected, so the kiosk displays the message “Please drive forward into the wash stall.”

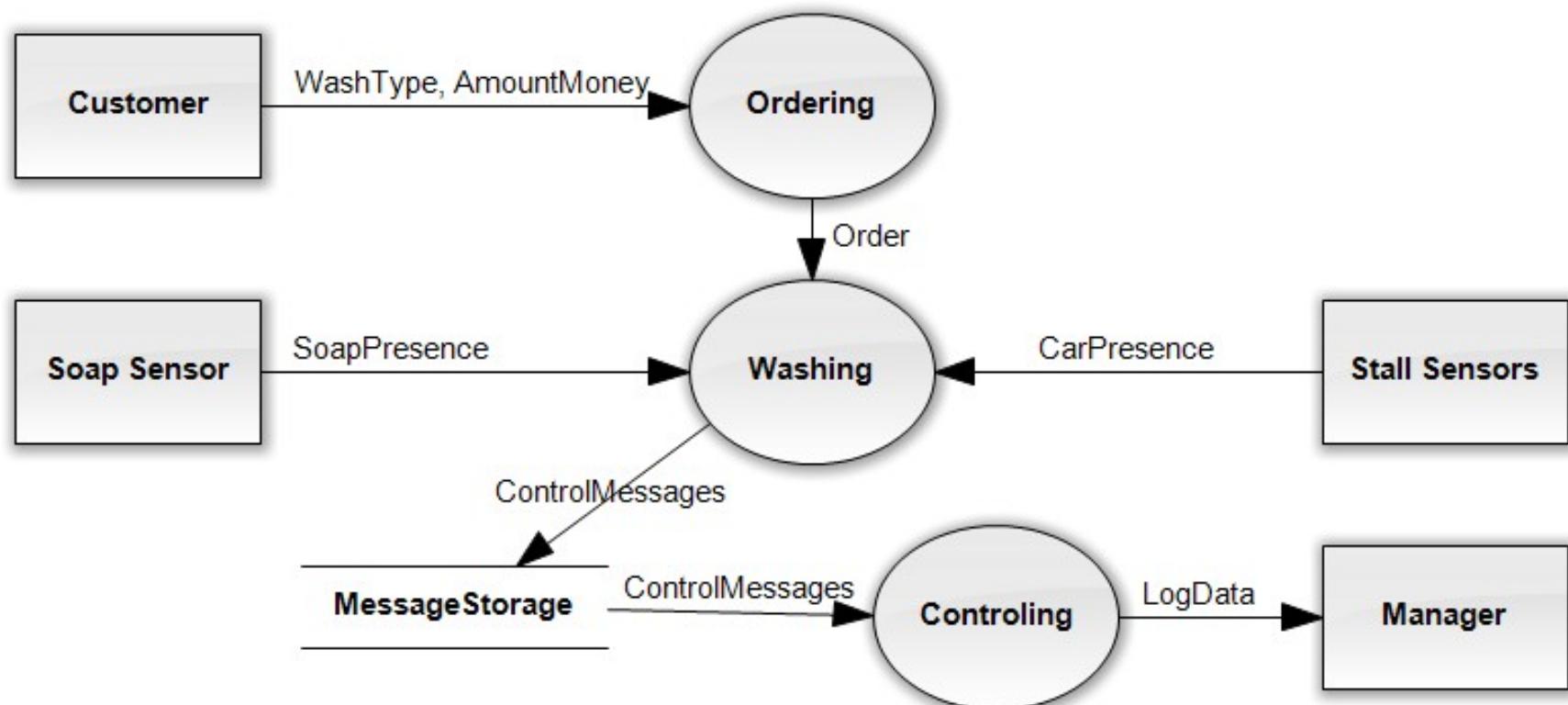
Division of a system into sub-processes

- ❖ In specifications the system is divided into subprocesses/components as seen from user perspective. DFDs can be used.
- ❖ Drawing of a contextual diagram:
 - ◆ External entities are actors in scenarios.
 - ◆ Define data flows from scenarios or event list.
- ❖ Drawing DFD level 0:
 - ◆ The system process is divided into sub-processes.
 - ◆ Each sub-process may be further divided...

Example – contextual diagram



Example – DFD level 0

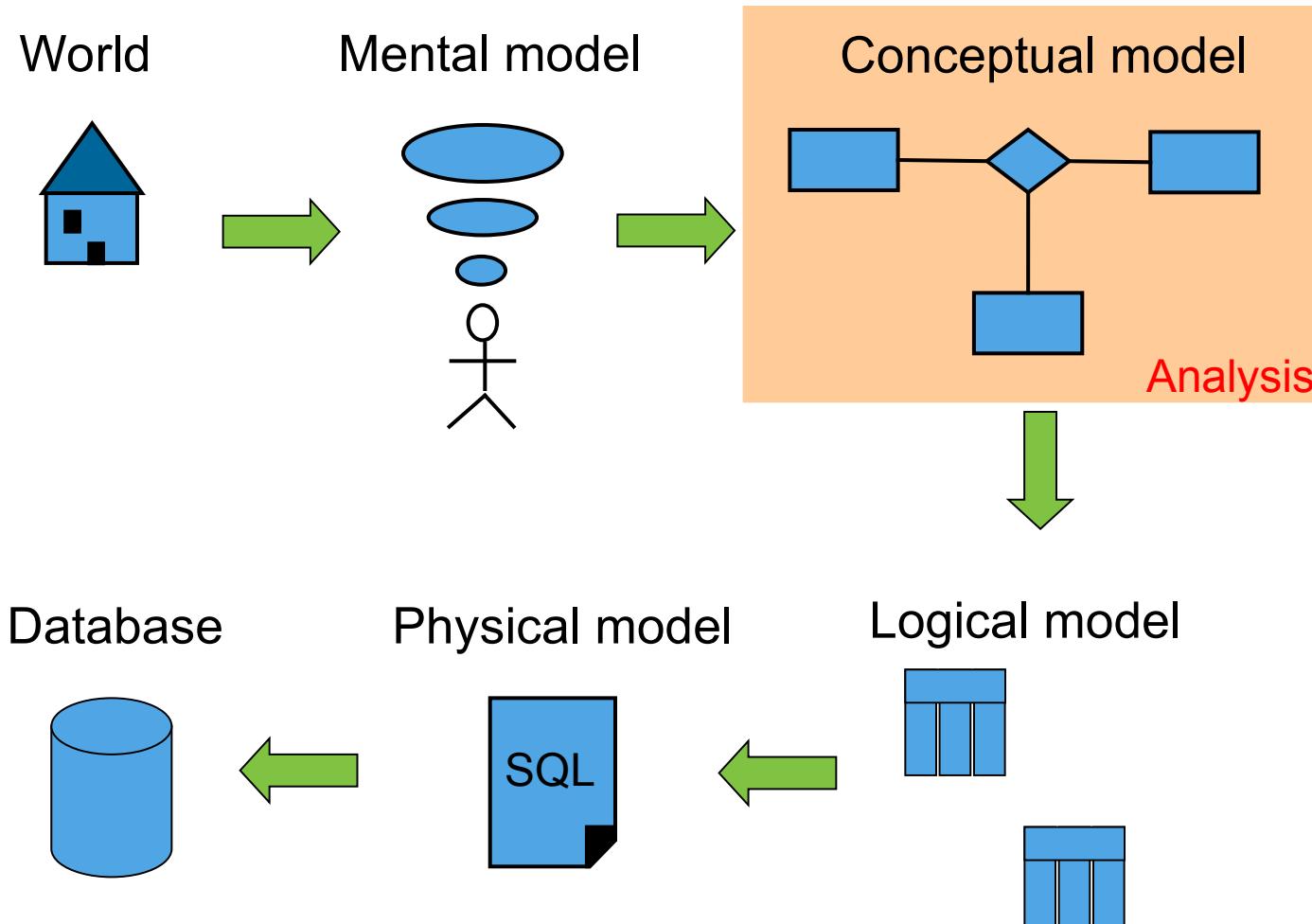


Data model

- ❖ Structural analysis typically uses
 - ◆ Data dictionary
 - ◆ Entity relational diagram (ER),
 - ◆ Relational data model
 - Typically defined later in a system or component design phase.
- ❖ Other models can also be used:
 - ◆ Entity life history (ELH)

Data model - procedure

From (SIS3-IS):



Object Oriented Analysis (OOA)

- ❖ More strict differentiation between analysis and design.
 - ◆ Still some variations possible.
- ❖ The basis is the usage of objects and presentation using object models.
 - typical usage of UML.

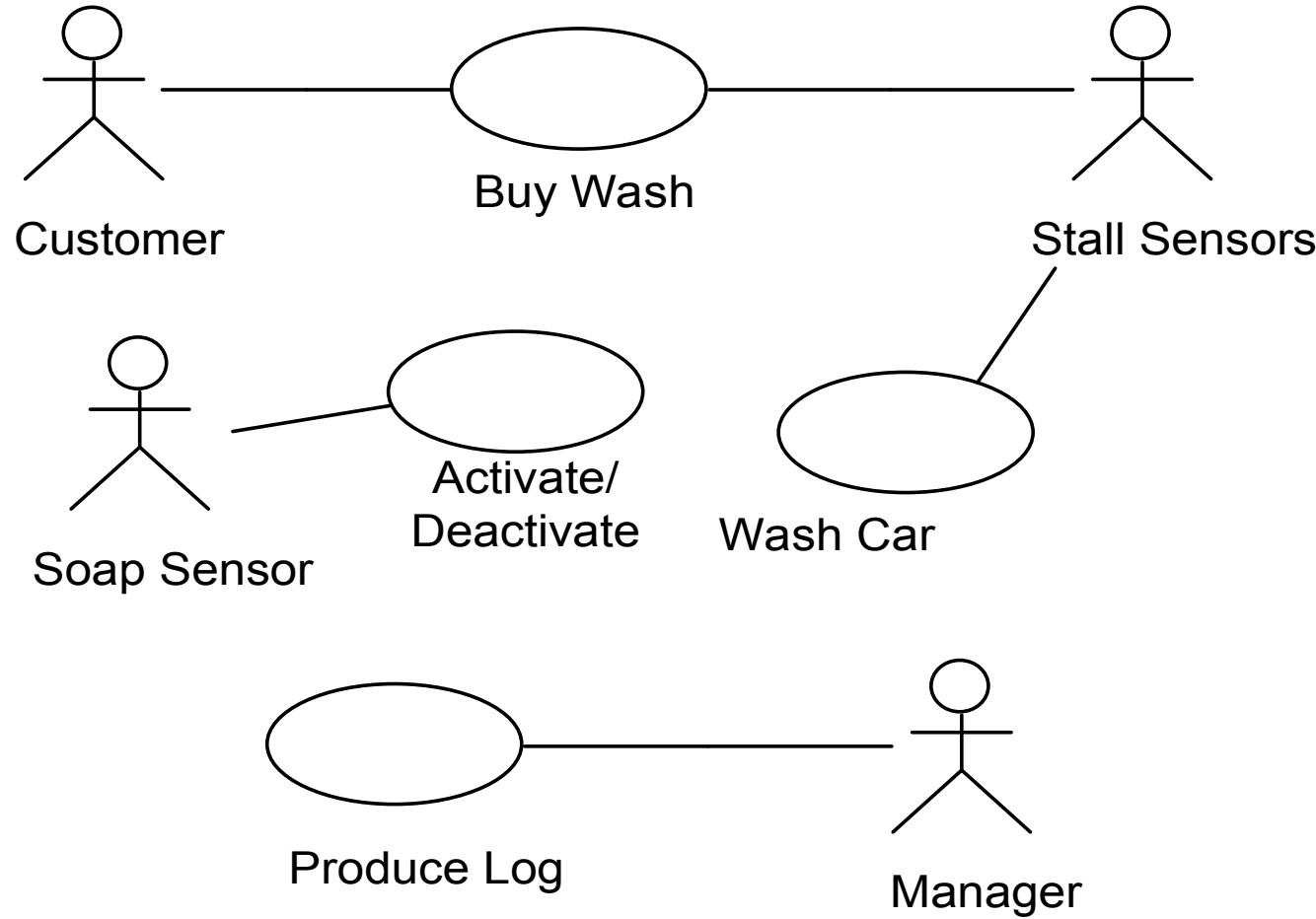
Use cases

- ❖ Use cases describe interaction between actors and a system for specific scenario of system usage:
 - ◆ Describe a service offered to user.
 - ◆ Describe interaction of user and system in details, all the processes involved from the beginning till the end of usage.
 - ◆ With a use-case users reach some goal.
 - ◆ Understanding of requirements is crucial for defining use cases.
 - ◆ Use cases are defined for each system vent.
 - ◆ Described in details using use-case descriptions and illustrated using use-case diagrams.

Use case models

- ❖ A use-case model consists of
 - ◆ A use-case diagram
 - ◆ A use-case description (for each use case shown in the diagram).
- ❖ Use case models do not enable a complete system description.
They must be complemented with additional requirements:
 - ◆ Data and nonfunctional requirements.
 - ◆ Physical layer requirements.

Use case diagram example



A use case diagram for a car wash

Use case description

- ❖ Use case description is specification of interaction between SW product and actor(s).
- ❖ Provide information of:
 - ◆ Activities of actor and activities of product,
 - ◆ All possible interaction courses

Use case description contents

1. Use case name
2. Actors
 - List of all the actors (people, external systems...)
3. Requirements
 - Requirements related to the use-case.
4. Trigger
 - The event that starts the activity (and preconditions)
5. Basic (main) flow
 - The sequence of user actions and system responses that will take place during execution of a use case and leads to accomplishing the goal.
6. Post conditions
 - Expected outcome of the use case with the state of the system at the conclusion of the use case execution.

Use case description contents

7. Alternative flows

- List and describe other legitimate usage scenarios that can take place with the use case.
- May start and end in anywhere related to other flows.
- May be based on other alternative flows.

8. Other

- Use case comments and links to other use cases.

Example

- ❖ Name: Activate/Deactivate
- ❖ Actor: Soap sensor
- ❖ Requirements:
 - Customers — Need their cars washed with soap, and want a complete wash
 - Operations — Want the carwash to operate without constant attention
- ❖ Trigger: One minute has passed since the last time the soap sensor was checked.
- ❖ Basic flow:
 1. The carwash queries the soap sensor.
 2. The soap sensor indicates that there is soap.
 3. If the carwash is active, it continues its operation and the use case ends.
- ❖ Postconditions:
 - ❖ The carwash is active if and only if the soap sensor indicates that there is soap.
 - ❖ No wash currently in progress is interrupted.

Example

- ❖ Alternative flows:
- ❖ 1a: The carwash is unable to query the soap sensor:
 - ❖ 1a1. The controller logs the problem and the use case ends.
- ❖ 2a: The soap sensor indicates that there is no more soap:
 - ❖ 2a1. If the carwash is inactive, the use case ends.
 - ❖ 2a2. If the carwash is active, the controller displays an out-of-order message and becomes inactive; the use case ends.
- ❖ 2b: The soap sensor does not respond:
 - ❖ 2b1. The controller logs the problem.
 - ❖ 2b2. If the carwash is inactive, the use case ends.
 - ❖ 2b3. If the carwash is active, the controller displays an out-of-order message and becomes inactive; the use case ends.
- ❖ 2a2, 2b3: A wash is in progress:
 - ❖ 2a21. The carwash completes the current wash, then displays an out-of-order message; the use case ends.
- ❖ 3a: The carwash is inactive:
 - ❖ 3a1. The carwash becomes active and displays a ready message.

Additional diagrams and models

- ❖ Use case flows can be documented using activity diagrams.
- ❖ System processes can be presented using DFD or communication diagrams or sequence diagrams.
- ❖ System data can be presented using ERD or class diagrams.
- ❖ Others – depending on the specific needs.

System decomposition

- ❖ OOA performs system decomposition as division on components, subcomponents and objects.
 - ◆ While structured approaches divide functionality – functional decomposition.
- ❖ Objects are defined with classes.
 - ◆ Selection of objects is based on a domain model (simple model without attributes and operations).

Modeling of classes

- ❖ The main tool for defining classes are use cases.
- ❖ One of the methods for defining classes is Noun Identification.
- ❖ Noun identification is simple but could be misleading (natural languages are not exact).

Noun identification

1. The **carwash** queries the **soap sensor**.
2. The **soap sensor** indicates that there is **soap**.
3. If the **carwash** is active, it continues its **operation** and the use case ends.
 - ❖ 1a: The **carwash** is unable to query the **soap sensor**:
 - ❖ 1a1. The **controller** logs the **problem** and the use case ends.
 - ❖ 2a: The **soap sensor** indicates that there is no more **soap**:
 - ❖ 2a1. If the **carwash** is inactive, the use case ends.
 - ❖ 2a2. If the **carwash** is active, the **controller** displays an out-of-order **message** and becomes inactive; the use case ends.
 - ❖ 2b: The **soap sensor** does not respond:
 - ❖ 2b1. The **controller** logs the **problem**.
 - ❖ 2b2. If the **carwash** is inactive, the use case ends.
 - ❖ 2b3. If the **carwash** is active, the **controller** displays an out-of-order **message** and becomes inactive; the use case ends.
 - ❖ 2a2, 2b3: A **wash** is in **progress**:
 - ❖ 2a21. The **carwash** completes the current **wash**, then displays an out-of-order **message**; the use case ends.
- ❖ 3a: The **carwash** is inactive:
 - ❖ 3a1. The carwash becomes active and displays a ready **message**.

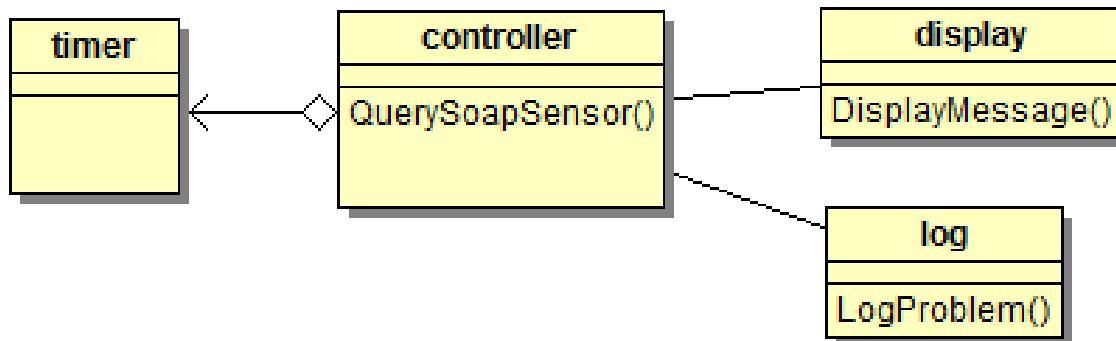
Selection of class candidates

- ❖ All nouns are not appropriate for classes:
 - ◆ Different nouns can have identical meaning
 - ◆ Nouns are not used only for classes but also for attributes
 - ◆ Some nouns do not represent objects of interest and are not needed.
- ❖ Searching for the simplest set of classes that satisfy requirements and offer good enough adaptability (considering maintenance).

Omitting criteria

- ❖ Redundant classes (nouns having the same meaning). Leave only the more informative one.
Example: customer/buyer.
- ❖ Unneeded classes – out of context.
- ❖ Unclear classes (indefinite). Example: collapse.
- ❖ Attributes: Examples: name, age, weight...
- ❖ Functions – description of class behavior. Example: acceleration.
- ❖ Roles – class names should show class nature not role.
Example: customer or car owner, car driver, car renter.
- ❖ Implementation constructs. Example: CPU, array, linked list, subroutine...

Example – class identification



Different classes groups may provide equivalently good solutions.
Each additional use-case can extend the list of eventual classes needed.

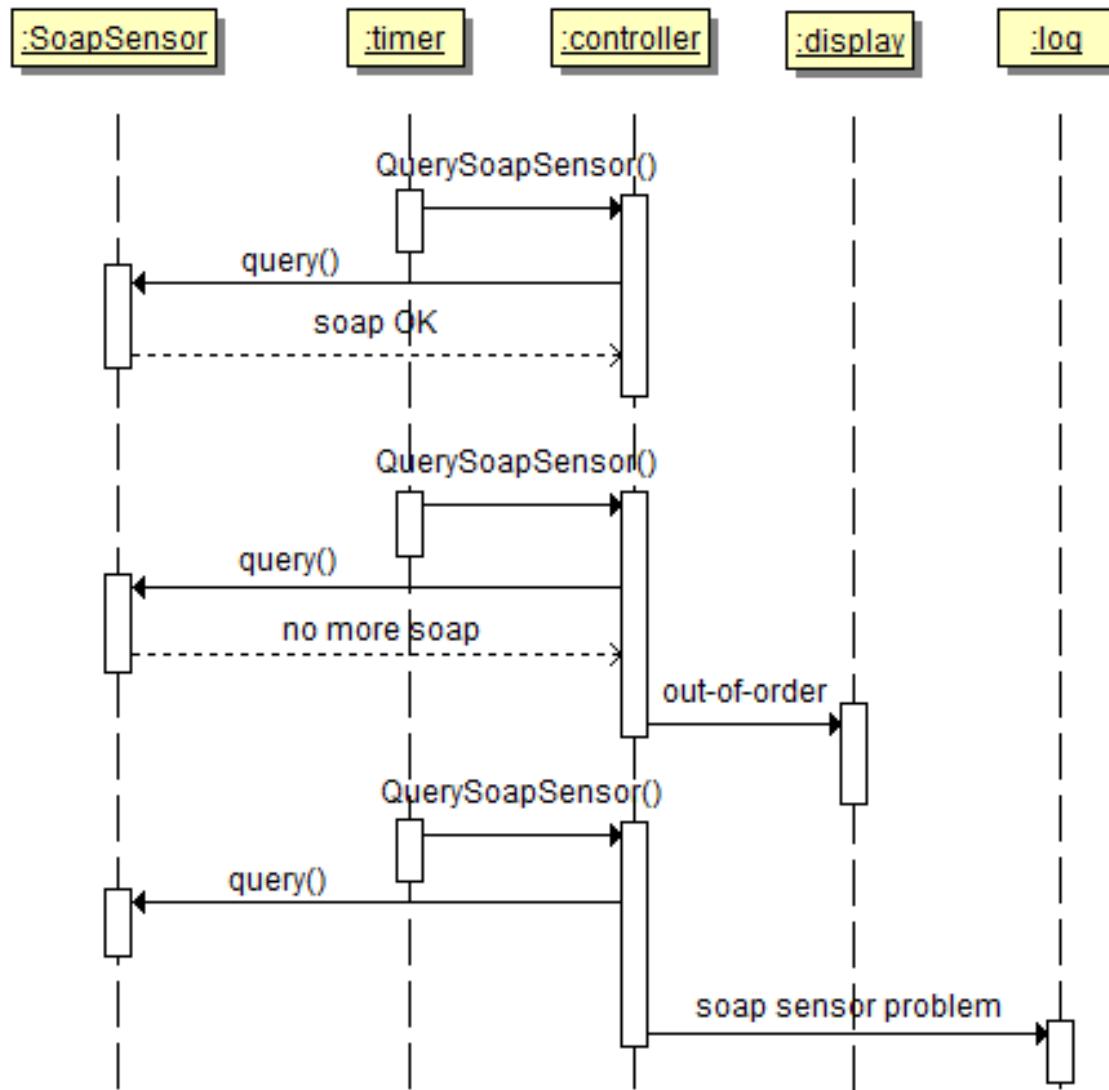
Dynamic models

- ❖ Class diagrams provide static view on classes. Their role and activities – the system dynamics can be shown using behavior diagrams.
 - ◆ Inter-class interaction:
 - UML sequence diagrams
 - UML communication diagrams
 - ◆ Intra-class activities:
 - UML state diagrams
 - UML activity diagrams

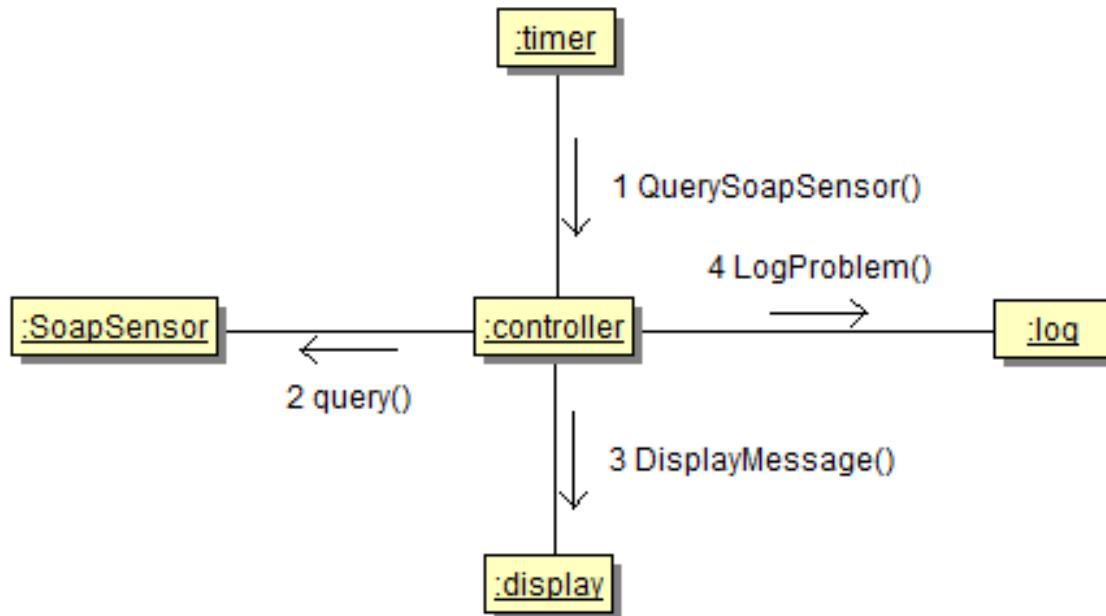
UML interaction diagrams

- ❖ Describe cooperation between system components (objects).
- ❖ Typically show behavior for individual use-case by showing objects and messages that they exchange.
- ❖ For us the most useful among interaction diagrams are sequence diagrams and communication diagrams. They provide the same information with different emphasis.
- ❖ Diagram selection is a matter of personal or organizational preferences.

Sequence diagram: example

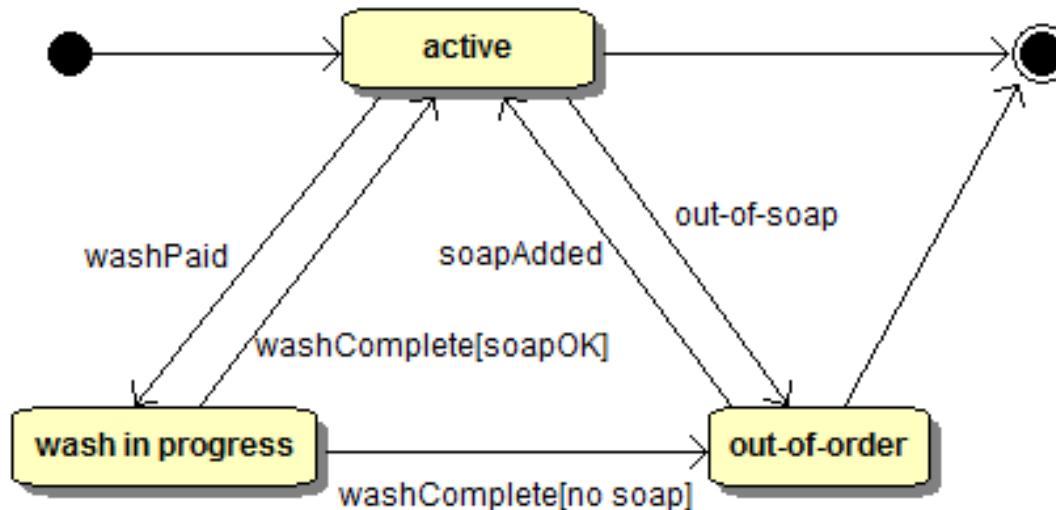


Communication diagram: example



State diagrams

- ❖ Describe an object life cycle.
 - ◆ All objects' states
 - ◆ State transitions as a consequence of events.
- ❖ Describe object behavior for all the use cases at once.

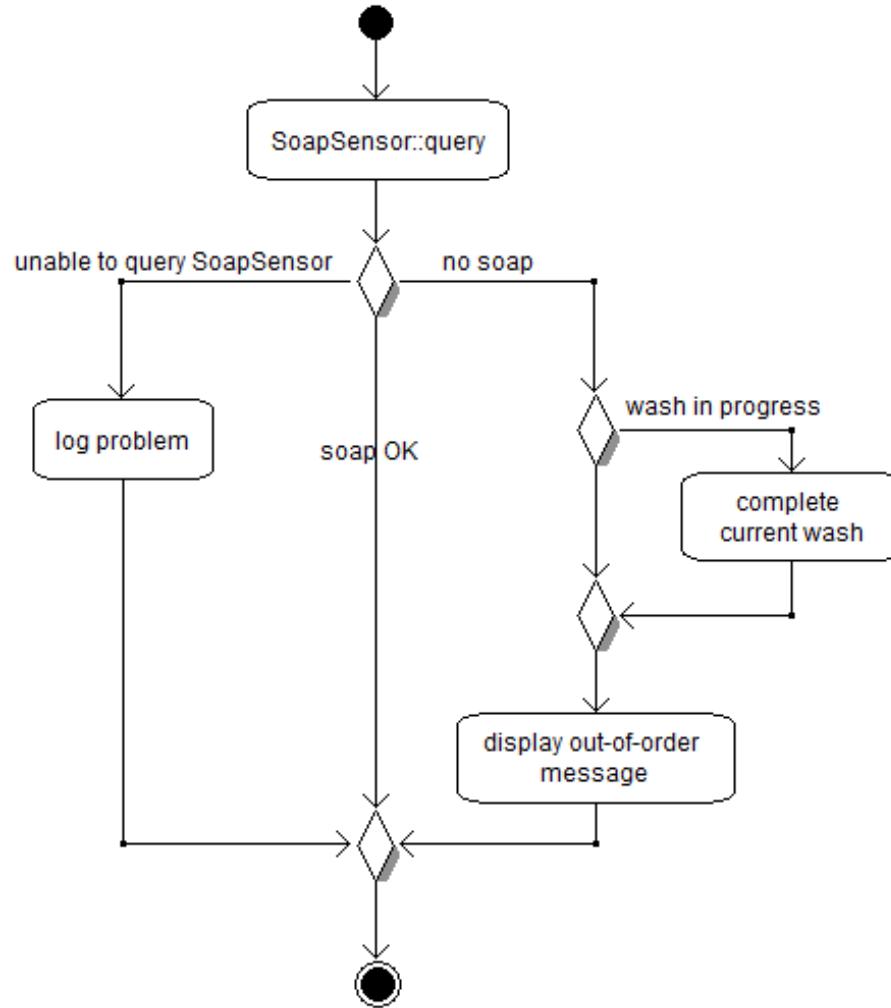


Example: A state diagram of object “controller”

Activity diagram (UML)

- ❖ UML activity diagram shows activity that is typically executed inside one class method, but could also be used for multiple components or a whole system.
- ❖ One activity can describe multiple use-cases.
- ❖ Enable to describe all the behavior of a system.

Activity diagram - example



After modeling...

- ❖ Not depending on the methodology used, different users declare high importance for different and potentially differently defined requirements!
 - Some coordination is needed to find the acceptable solution.

Requirements coordination

- ❖ Customers/users often wish more than can be achieved (mostly due to cost limits).
- ❖ Different stakeholders provide different requirements that may be incompatible or contradictory.
 - ◆ Example: management limits costs or time, while users wish additional functionality or higher system performance / throughput.
 - ◆ Each stakeholder can declare his requirements with the highest importance.
- ❖ System analyst must resolve the discrepancies with negotiations.
 - ◆ Define system priorities.
 - ◆ Identify risks related to requirements.
 - ◆ Briefly estimate (“guesstimate”) required development cost for each of the requirements.
 - ◆ Discuss requirements supplemented with that data with stakeholders.
- ❖ Coordination is an iterative process of modifying, grouping, omitting requirements to reach a consensus.

The SRS document

❖ Possible contents of SRS:

- ◆ Introduction (goals, scope, definitions, references on other documents)
- ◆ General description (relation to other systems, functionality overview, definition of users and their specifics, general limitations)
- ◆ Requirements for each of the component (description, inputs, processing, outputs)
- ◆ Requirements of external interfaces (data formats, HW, relations to other SW or systems)
- ◆ Performance requirements
- ◆ Design limitations (standards, HW limitations...)
- ◆ Other requirements

❖ The contents shall depend on project needs.

SRS templates

- ❖ Search internet (do not trust everything)
- ❖ Wikipedia...
http://en.wikipedia.org/wiki/Requirements_analysis
- ❖ IEEE 830-1998 (Recommended Practice for Software Requirements Specifications)
- ❖ A later standard:
ISO/IEC/IEEE 29148:2011

Recension

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Recension

- ❖ Recension is an examination and assessment of a product or a process from a qualified individual or group.
- ❖ Recension of SRS shall be made by stakeholders (customers, management...).

SRS recension

❖ Review of system requirements (all):

- ◆ Validity. Does the system offer functions/services that best fulfill user requirements?
- ◆ Consistency. Are any of the requirements opposing each other?
- ◆ Completeness. Does the system includes all the functions required by users?
- ◆ Reality. Can the requirements be fulfilled with the proposed technology with the limited resources (cost).

❖ Review of individual requirements:

- ◆ Testability. Is it possible to verify the compliance with the requirement?
- ◆ Understandability. Is there only a single understanding of the requirement possible?
- ◆ Tracability. Is the source of the requirement clearly defined?
- ◆ Adaptability. Can the requirement be changed without big influence on other requirements?

Recension types

- ❖ Control from an individual
 - ◆ Often performed by the author himself.
 - ◆ Using **control lists**.
- ❖ Group review
 - ◆ By a group of reviewers.
 - ◆ The procedure is not predefined and the roles are not defined. v
 - ◆ Usually each reviewer previously make a review as an individual using a control list and then combine the results.
- ❖ Supervision
 - ◆ Formal recension by a group of trained supervisors with defined roles.
 - ◆ Use of control lists
 - ◆ Supervisors must prepare for the supervision.

SRS supervision

- ❖ The goal is to find as many errors and risks as possible.
- ❖ Not intended to judge the author.
- ❖ Supervision does not correct errors.
- ❖ Supervision is expensive and long-lasting.
- ❖ Supervision is effective.

An example of a control list

- Every requirement is atomic.
- Every requirement statement uses “must” or “shall.”
- Every requirement statement is in the active voice.
- Terms are used with the same meaning throughout.
- No synonyms are used.
- Every requirement statement is clear.
- No requirement is inconsistent with any other requirement.
- No needed feature, function, or capability is unspecified.
- No needed characteristic or property is unspecified.
- All design elements are specified to the physical level of abstraction.
- Every requirement is verifiable.
- Similar design elements are treated similarly.
- Every requirement can be realized in software.
- Every requirement plays a part in satisfying some stakeholder’s needs or desires.
- Every requirement correctly reflects some stakeholder’s needs or desires.
- Every requirement statement is prioritized.
- Every requirement priority is correct.

Conclusions of a supervision

- ❖ The moderator concludes that
 - ◆ Errors exist and need to be corrected or
 - ◆ No errors are found.
- ❖ Author corrects the errors found by supervisors.
- ❖ In the case of larger changes and in the case of high probability of additional errors another supervision needs be made (repeated).

Requirements management

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Requirements management

“...is the process of documenting, analyzing, tracing, prioritizing and agreeing on requirements and then controlling change and communicating to relevant stakeholders. It is a continuous process throughout a project. “

Src: https://en.wikipedia.org/wiki/Requirements_management

Requirements management

Changes of requirements

- ◆ are unavoidable and
- ◆ happen during the whole life cycle.
- ❖ Requirements management include
 - ◆ Identification,
 - ◆ Tracing
 - ◆ Prioritizing
 - ◆ Validating
 - ◆ Communicating with stakeholders

BENEFITS OF Requirements Management Tools

Remove

-  Ambiguity
-  Assumptions
-  Wishful thinking
-  Gray Area
-  Interpretations

Ensure your implementation and deliverables are:

-  Clear
-  Realistic
-  Agreed-upon

Read the full article at: thedigitalprojectmanager.com/requirements-management-tools



Conclusion

Analysis requires precision ;-)

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Different possible interpretations



Eskimo or Indian?

May be misunderstood!



Vrečka.

Designed to be misunderstood...



Nuns sitting.

Unnoticed details?



A face in a coffee

Software engineering

System design

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

An architectural design example

❖ Designing the house:

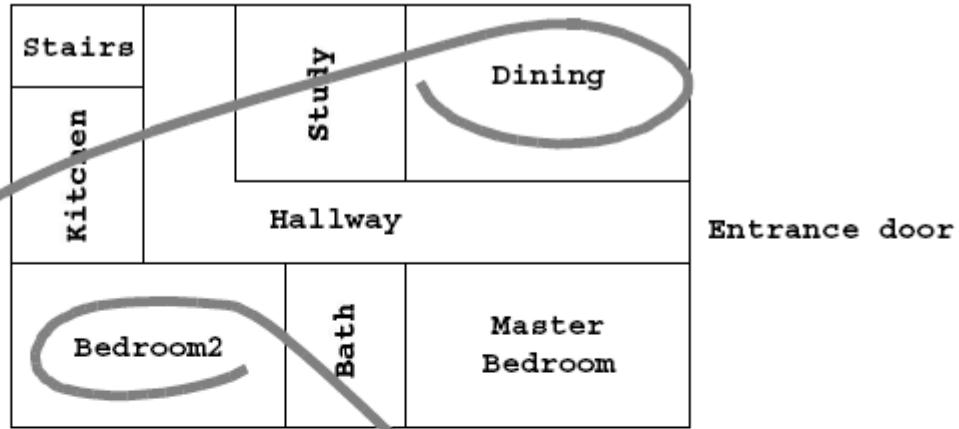
- ◆ Starts after getting an agreement with the customer regarding the number of rooms, floors, size and location of the house.
- ◆ The goal is to set a floor plan that defines the arrangement of rooms/ walls, position of windows and doors, heating, water pipes, drain.
- ◆ The architect must consider several standards (standard (kitchen) element/cabinet dimensions, standard bed sizes...).
- ◆ The architect must not limit the actual furnishing!

❖ Architectural design starting point (SRS):

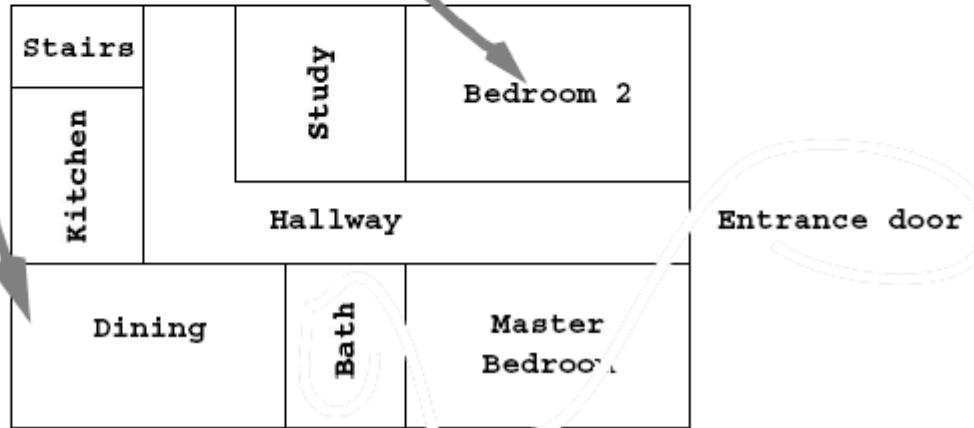
1. The house must have two bedrooms, study room, kitchen, living room... (functional requirements).
2. The distance walked by the residents shall be as small as possible (nonfunctional requirement).
3. Maximum use of daylight (non-functional requirement).
4. Best use of space (non-functional requirement).

Architectural example

Initial version
N



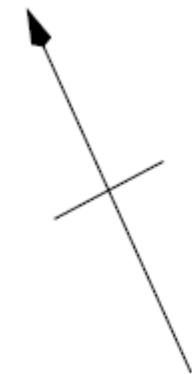
Second version



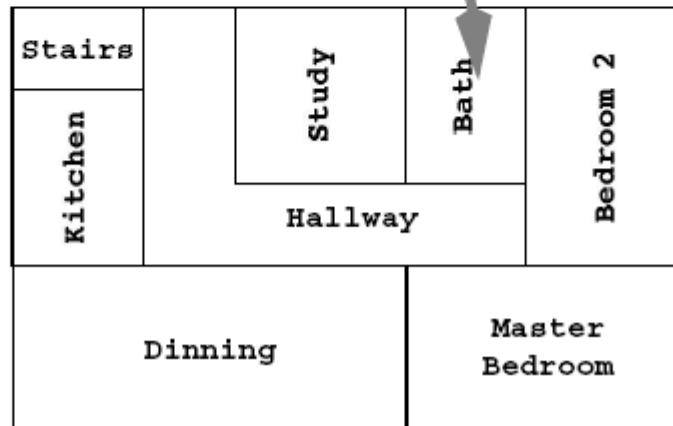
Architectural example

Second version

N



Third version



Architectural example

- ❖ Similarities with software architecture design:
 - ◆ The whole system gets devided into less complex components (house → rooms == system → components (subsystems)).
 - ◆ Connections between components are important (doors, corridors == interfaces)
 - ◆ Non-functional requirements (room size == performance).
 - ◆ Functional requirements (required rooms == use cases).
 - ◆ System design influences component design and implementation (kitchen plan == component design).
 - ◆ Difficulty (and price) of later changes (moving the walls == changing interfaces).
 - ◆ Component design is left for further design stages (furnishing plan == component design).

System design

- ❖ System design is activity of translating a SRS into a system design model.
- ❖ **The System Design Model divides the system into subsystems (components), describe their behavior, their relationship and interfaces.**
- ❖ It includes all the decisions that influence on structure of the whole system also known as **Software Architecture**.
- ❖ System design is not algorithmic!

Stable architecture

- ❖ A stable architecture assures reliability of a system and “easy” maintenance.
- ❖ Stable architecture enables adding of functionalities with minimal architecture changes.

System design

- ❖ System design includes:
 - ◆ Definition of design goals.
 - ◆ Decomposition of a system into subsystems.
 - ◆ Selection of (already developed) reusable components.
 - ◆ Relation between SW and HW.
 - ◆ Selection of infrastructure for persistent data management.
 - ◆ Definition of the access control policy.
 - ◆ Selection of a global control flow mechanism.
 - ◆ Handling borderline cases.

Definition of design goals

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Design goals

- ❖ Definition of design goals is the first step of system design.
- ❖ Most design goals follow the nonfunctional requirements or application domain.
- ❖ Other design goals must be coordinated with the customer.
- ❖ In general the design goals can be selected from a predefined list of general SW product quality wishes that could be grouped in five categories:
 - ◆ performance,
 - ◆ dependability,
 - ◆ cost,
 - ◆ maintenance,
 - ◆ user criteria.

Design goals - performance

- ❖ Response time
 - ◆ Time between request and response.
- ❖ Throughput
 - ◆ Number of tasks executed in a time unit.
- ❖ Memory
 - ◆ How many memory (or storage in general) is required for normal operation?

Design goals - dependability

- ❖ Robustness
 - ◆ Ability to survive irregular user request (input flow).
- ❖ Reliability
 - ◆ Difference between specified and observed behavior.
- ❖ Availability
 - ◆ Percentage of time when system can be used for user tasks.
- ❖ Fault tolerance
 - ◆ Ability to operate under erroneous conditions.
- ❖ Security
 - ◆ Ability to resist malicious attacks.
- ❖ Safety
 - ◆ Ability to not endanger human lives, even in the case of errors and defects.

Design goals - cost

- ❖ **Development cost**
 - ◆ Cost of development of the (initial) system.
- ❖ **Deployment cost**
 - ◆ Cost of installation and user education.
- ❖ **Upgrade cost**
 - ◆ Cost of data translation from previous systems.
- ❖ **Maintenance cost**
 - ◆ Cost of error correction and system extensions.
- ❖ **Administration cost**
 - ◆ Cost of system administration.

Design goals – maintenance

- ❖ Extensibility
 - ◆ How difficult is to implement additional functionality.
- ❖ Modifiability
 - ◆ How difficult is to change existing functionality.
- ❖ Adaptability
 - ◆ How difficult is to transfer the system to other user domains.
- ❖ Portability
 - How difficult is to transfer the system to other platforms?
- ❖ Readability
 - ◆ How difficult is to understand the system by reading the source code.
- ❖ Traceability of requirements
 - ◆ How difficult is to link the code with specific requirement.

Design goals – user criteria

- ❖ Utility
 - ◆ How good the system support user's work?
- ❖ Usability
 - ◆ How difficult is to use the system (by users)?

Design goals contradictions

- ❖ At the same time only a part of all criteria can be satisfied.
 - ◆ Example: it is not realistic to expect a system that is secure and inexpensive.
- ❖ The developer typically defines priorities for the design goals and use them to ease the decisions during system development.

Opposing design goals

- ❖ **Memory versus throughput**
 - ◆ Throughput and response time can be increased at the expense of memory usage.
- ❖ **Delivery time versus functionality**
 - ◆ Faster delivery times can be ensured by limiting system functionality.
- ❖ **Delivery time against quality**
 - ◆ In the absence of time, the decision may be made to hand over a product with known defects.
- ❖ **Delivery time vs. number of developers**
 - ◆ Shortening delivery times by increasing the number of developers is only reasonable in the initial stages of development, otherwise, by increasing the number of developers, the development time can even be extended (due to need for training, extensive communication...)
- ❖ ...

SW system decomposition

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

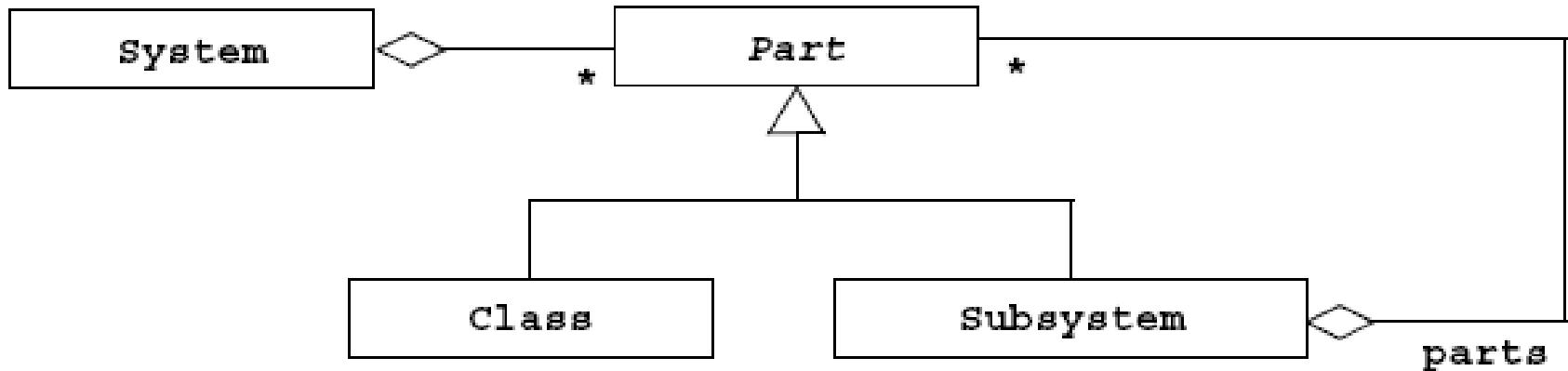
“Divide and conquer”

- ❖ Dealing with large systems is much more difficult than dealing with a series of small components - subsystems.
- ❖ Divide and conquer!
 - ◆ Smaller components are easier to understand than large ones.
 - ◆ Each subsystem can be taken care of by other people.
 - ◆ Possibility of specialization of individual software engineers.
 - ◆ Components can be (later) replaced without major repairs to the rest of the system.

Identification of subsystems

- ❖ Identifying subsystems requires some inventiveness from the system designer (like finding objects in the specification stage of a requirements analysis).
- ❖ Each additional request may:
 - ◆ combine several subsystems into one subsystem,
 - ◆ divide complex subsystems into several subsystems,
 - ◆ adds subsystems that provide additional functionality.
- ❖ Possible heuristic approach:
 - ◆ Objects from the same use case should be part of the same subsystem.
 - ◆ A separate subsystem should be included to transfer data between subsystems.
 - ◆ Minimize the number of connections between subsystems.
 - ◆ Keep all objects in the subsystem functionally connected.

The concept of subsystems



The concept of subsystems

- ❖ Subsystems provide services to other subsystems.
- ❖ The **service** is a group of related operations.
- ❖ When designing a system (architecture), it is necessary to define subsystems in terms of the services that subsystems provide.
- ❖ The subsystem operations available to other systems form the subsystem interface, also called the application programmer interface (API).

Service planning

- ❖ When designing the system, the services of each subsystem must be defined:
 - ◆ list operations,
 - ◆ determine their parameters,
 - ◆ determine their high-level behavior.
- ❖ Only later during component design the API is updated with:
 - ◆ type of parameters
 - ◆ the types of data returned by each operation

Decomposition of software systems

- ❖ The distributed system is divided into clients and servers.
- ❖ The system is subdivided into subsystems.
- ❖ The subsystem may consist of one or more packages.
- ❖ The package consists of classes.
- ❖ The class contains several methods.

Layers and partitions

- ❖ **Layers** separate the subsystems hierarchically, vertically.
- ❖ **Partitions** separate subsystems of the same layer, horizontally.
- ❖ Each subsystem (layer or partition) is responsible for different services. The subsystems are weakly interconnected and often operate independently.

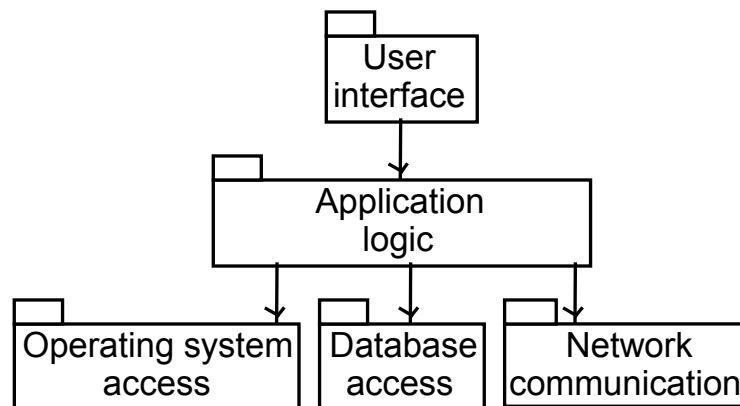
Layers

- ❖ The layers form a hierarchical structure. The layer provides services to the upper layers and depends only on the lower layers:
 - ◆ Closed architecture: The layer uses only the services of the first lower layer.
 - ◆ Open architecture: The layer can use the services of all the lower layers.
- ❖ A layer can be replaced without affecting other layers.

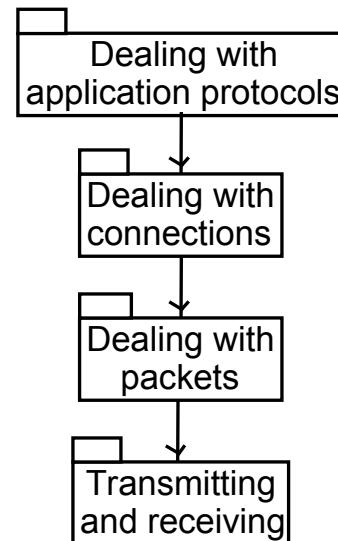
Layers

- ❖ Layers in complex systems provide insights into the system at different levels of abstraction (and thus levels of abstraction in system development):
 - ◆ It is important to have a separate layer dedicated to the user interface (UI).
 - ◆ The layer under UI provides functionality defined by use-cases.
 - ◆ The lowest layers provide general services, e.g. network communications, database access...

Layers example

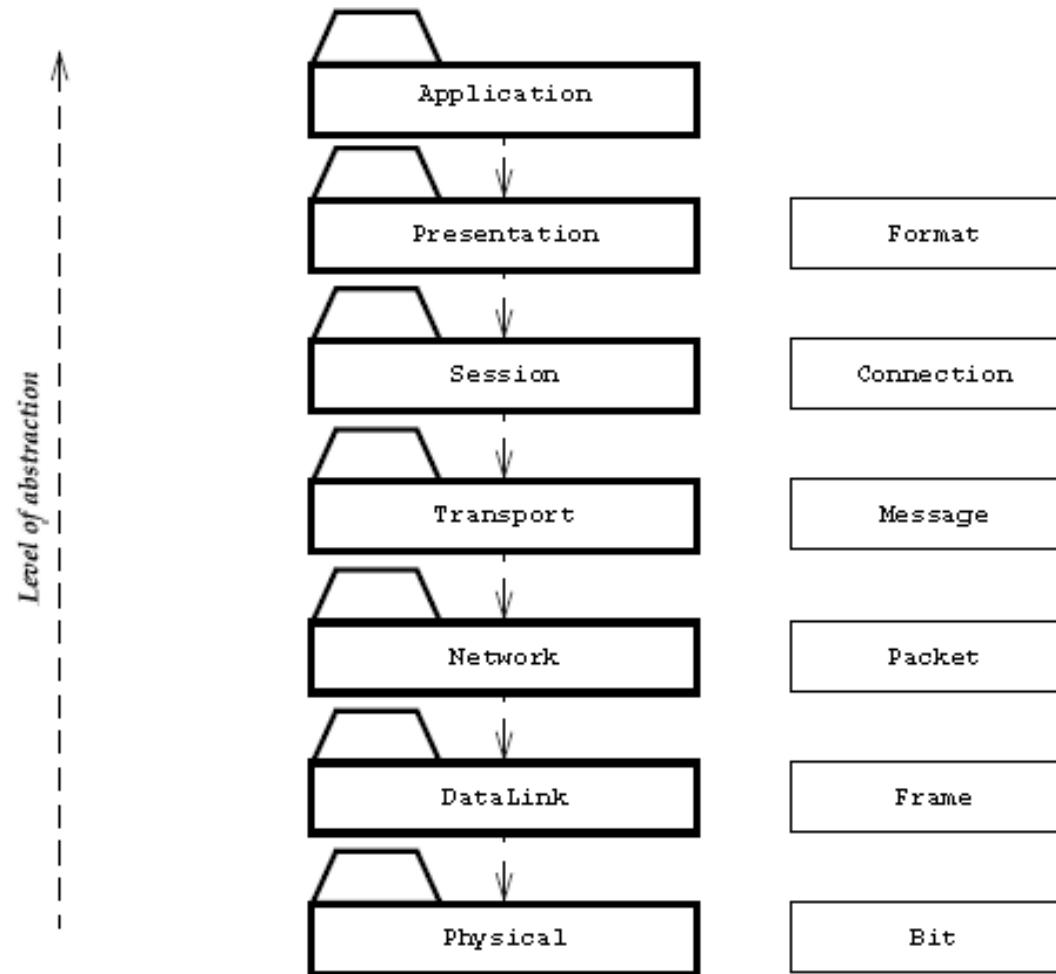


a) Typical layers in an application program



c) Simplified view of layers in a communication system

Layers of the ISO OSI model



Subsystem coupling

- ❖ Coupling refers to the relationship between two subsystems.
- ❖ If the two subsystems are strongly interconnected, changes to one subsystem will have an impact on the operation of the other subsystem.
- ❖ It is desirable that the subsystems are connected as weakly as possible.
 - ◆ This reduces the impact of errors and changes of one subsystem on the performance of another subsystem.
- ❖ There are several types of coupling:
 - ◆ Content, Common, Control, Stamp, Data, Routine Call, Type use, Inclusion/Import, External

Content coupling

- ❖ Content coupling - when one component secretly changes the internal data of another component.
 - ◆ To reduce content coupling, it is suggested to encapsulate all instance variables:
 - defining variables as private,
 - providing get and set methods.

Common coupling

- ❖ Common coupling - when global variables are used
 - ◆ All components that use a global variable become interconnected.
 - ◆ They can be acceptable if the global variables represent the system default values.
 - ◆ For global access to an object, a “Singleton” design pattern may be used.

Control coupling

- ❖ Control coupling - if one procedure calls another procedure, specifying a command that explicitly defines the operation of the other procedure.
 - ◆ In case of changes, both procedures need to be changed.
 - ◆ Control connectivity can often be avoided by using polymorphic operations (the same operation may be performed differently for different input types/classes).

Stamp coupling

- ❖ Stamp coupling - when one of the classes is used as an argument type of a method (in the other class).
 - ◆ Because one class uses the other, system changes are more difficult:
 - changes are required for both classes,
 - reusing a class requires reusing both classes.
 - ◆ There are two ways to eliminate this kind of coupling:
 - use of interface for an argument type.
 - use of simple variables (primitive / simple types).

Data coupling

- ❖ Data coupling - when argument types of methods are primitive or simple library classes.
 - ◆ Calling a method from another class requires (multiple) data to be provided. The more arguments there are, the greater the coupling.
 - ◆ Coupling can be reduced by not using unnecessary arguments.
 - ◆ Data coupling is related to stamp coupling: Reducing one typically increases the other.

Routine call coupling

- ❖ Routine call coupling - when one routine (or another method in the OO system) calls another.
 - ◆ Routines are coupled because one routine depends on the behavior of the other routine.
 - ◆ Routine connectivity is always present, in every system.
 - ◆ If a sequence of two or more methods (routines) is used repeatedly, routine coupling can be reduced by constructing a routine that encapsulates the sequence.

Type use coupling

- ❖ Type use coupling - when a module uses the data type specified in another module.
 - ◆ Using the class of the second module as the data type for the declaration of variables.
 - ◆ In the case of changing the definition of a data type, this may also require a change in the procedures that use this type.
 - ◆ For a OO system, the most general class (or interface) containing the required operations should be used.

Inclusion or import coupling

- ❖ Inclusion or import coupling - when one component includes a package ("import" in Java) or another component ("include" in C++)
 - ◆ A component is subject to everything that is contained in the included component.
 - ◆ Changes or additions to the included component can lead to conflicts (conflicts in variable names, functions, definitions ...).

External coupling

- ❖ External coupling - where the module depends on the operating system, shared libraries or hardware.
 - ◆ The number of places where such dependencies occur should be kept to a minimum.
 - ◆ External coupling can be reduced by using a “Façade” design pattern.

Cohesion of subsystems

- ❖ Cohesion refers to the internal relations inside a subsystem.
- ❖ Cohesion is high if the subsystem combines the interdependent or linked components and does not contain other, unrelated components.
- ❖ Greater cohesion of the subsystem is desirable, because such subsystems are easier to understand and modify.
- ❖ There are several types of cohesion:
 - ◆ Functional, Layer, Communicational, Sequential, Procedural, Temporal, Utility

Functional cohesion

- ❖ Functional cohesion - is obtained when all the code that is intended to produce a specific result is combined, while the rest of the code is not included.
 - ◆ When the module is intended to perform a single operation.
 - ◆ Advantages for the system:
 - easier to understand
 - increased reusability
 - easier component replacement

Layer cohesion

- ❖ Layer cohesion - the layer includes all that is needed to provide or access a related services (and everything else is not included).

Communicational cohesion

- ❖ Communicational cohesion - all modules that access or use certain data are grouped together (eg into one class), everything else is not included.
 - ◆ The class has good communication cohesion:
 - if it includes all system operations associated with storing and managing its data.
 - if it does not perform operations unrelated to the management of its data.
 - ◆ The main advantage: When data-related changes are required, all relevant code is located in one place.

Sequential cohesion

- ❖ Sequential cohesion – grouping procedures that follow one after the other - one procedure provides input for the other procedure, all the rest is not included.
 - ◆ The prerequisite for ensuring sequential cohesion is to satisfy all of the previously mentioned cohesion types.

Procedural cohesion

- ❖ Procedural cohesion - the procedures that are used one after the other are combined.
 - ◆ Even if one procedure is not related by the inputs/outputs (one procedure does not provide the input to the other).
 - ◆ Procedural cohesion is weaker than sequential.

Temporal cohesion

- ❖ Temporal cohesion - operations that are executed at the same stage of software execution are combined, everything else is not included.
 - ◆ Example: A code running during system initialization or shutdown grouped.
 - ◆ Temporal cohesion is weaker than the procedural one.

Utility cohesion

- ❖ Utility cohesion - operations that cannot logically be integrated into other cohesive units are grouped.
 - ◆ A utility is a procedure or class that is more widely used across multiple subsystems and is designed to be reusable.
 - ◆ Example: class `java.lang.Math`

Dependency models

- ❖ Dependency models help to analyze software architectures in terms of identifying problematic dependencies between modules.
- ❖ There are two popular ways to look at module dependencies:
 - ◆ Dependency graph
 - ◆ Dependency matrix

Dependency (structure) matrix

\$root		μ	ν	ω	δ
Module A	1	.		1	2
Module B	2		.	2	
Module C	3	4		.	3
Module D	4				.

Figure 1A

\$root		μ	ν	ω	δ
Module D	1	.			
Module A	2	2	.	1	
Module C	3	3	4	.	
Module B	4			2	.

Figure 1B

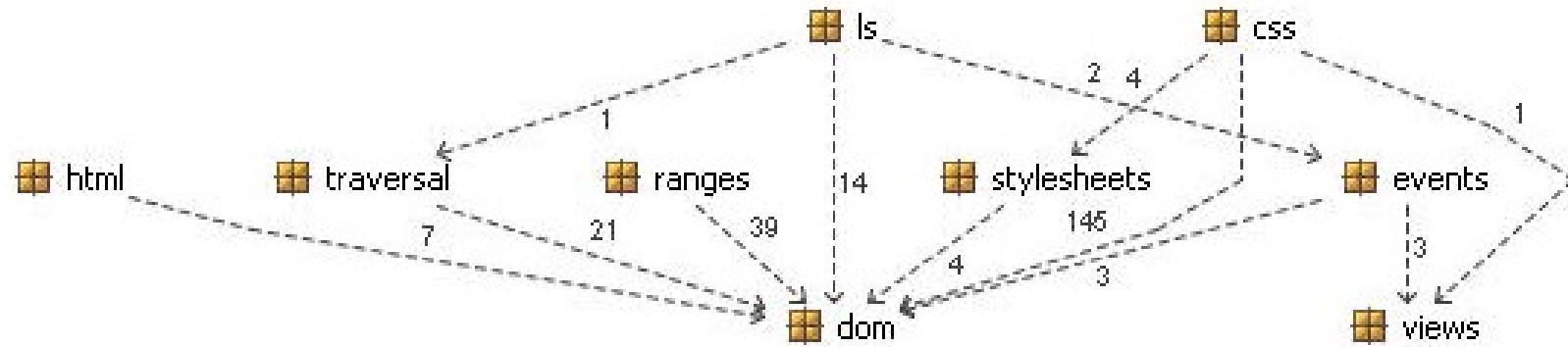
DSM

- ❖ We wish the lower triangular dependency matrix, which indicates that there are no cyclic dependencies between modules in the system.
- ❖ The example shows a cyclic dependency between modules A and C. If combined, the system has no cyclic dependencies.

See: https://en.wikipedia.org/wiki/Design_structure_matrix

Dependency graph

- ❖ The diagram shows the modules and their relationships.



See: https://en.wikipedia.org/wiki/Dependency_graph

CASE tools:

<http://www.headwaysoftware.com/products/structure101/index.php>

<http://www.coverity.com/products/architecture-analysis.html>

...

Relation between SW and HW

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Relation between subsystems (HW, SW)

- ❖ Many systems run on multiple computers that communicate with each other over an intranet or the Internet.
- ❖ Distribution of system components across computers (nodes) and communication infrastructure plan is required.
- ❖ Often, additional subsystems are required to divide subsystems by nodes:
 - ◆ Communication subsystems designed to transmit data and ensure transaction concurrency.
 - ◆ Control systems designed to ensure reliability.

Relation between subsystems (HW, SW)

- ❖ By arranging subsystems across nodes, functionality and processing power can be distributed to be available where we need it / or to be used where available.
- ❖ The challenges of storing, transferring duplication and synchronizing data.

Persistent data management

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Persistent data stores

- ❖ Persistent data is not lost when ending / closing the software.
- ❖ Where and how the data is stored affects the decomposition of the system.
 - ◆ In some cases (the "shared repository" architecture example) the entire subsystem is intended for this purpose.
- ❖ The first step in data storage planning is to identify persistent data, followed by a decision on how to store it:
 - ◆ What data should be persistent?
 - ◆ How to access persistent data?

Persistent data management

- ❖ Persistent data management is often a bottleneck for system performance:
 - ◆ Access to data must be fast and reliable.
 - ◆ It often requires the selection of a database management system or additional subsystems dedicated to persistent data management.

Persistent data storage

❖ Options:

- ◆ Direct use of a file system (flat files).
 - Access to data at a relatively low level.
 - The application itself has to solve the problems of simultaneous access to data, data loss due to downtime ...
- ◆ Relational databases
 - The level of abstraction already ensured in accessing the data.
 - May be a performance bottleneck.
- ◆ Object-oriented databases
 - Higher level of abstraction. They store data in the form of objects and links.
 - Typically slower performance than relational bases.

Persistent data storage

- ❖ When to choose a file system?
 - ◆ Large-scale data (e.g., images),
 - ◆ Provisional data (e.g. core files),
 - ◆ Low density of information (eg archive files, logs).
- ❖ When to choose a database?
 - ◆ Simultaneous access,
 - ◆ Access to some details only,
 - ◆ Multiple platforms used,
 - ◆ Multiple applications use the same data.
- ❖ When to choose relational databases?
 - ◆ Complex attribute queries.
 - ◆ Large databases.
- ❖ Object oriented databases?
 - ◆ Extensive use of links between data acquisition objects.

Access control policy

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Access control policy

- ❖ In multi-user systems, different users may have access to different data (authorization issue).
- ❖ It is necessary to determine how users in the system authenticate themselves (prove authenticity).
- ❖ In general, it is necessary for each actor to determine the operations to which he has access and to which of the data (objects).
- ❖ Access control is a system operation. It must be common and uniform for all subsystems.

Access control policy

❖ Possible solutions:

- ◆ Global access table
 - Specifies the rights for the relation (actor, object, operation). If the rights are not explicitly granted, access is denied.
- ◆ Access control list
 - For each class, it determines access rights for the relation (actor, operation). Example: guest list at a party.
- ◆ Capability
 - It connects the actor with the relation (class, operation). Allows the user with certain abilities to access the object. Example: entering a party with an invitation.

A global control flow mechanism

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Global control flow

- ❖ Questions:
 - ◆ How is the order of operations or activities determined in the system?
 - ◆ Does it allow more than one concurrent user interaction?
 - ◆ How the system expects external (user) interaction.
- ❖ The choice of a control flow mechanism affects the interfaces of the subsystems:
 - ◆ Event-driven control flow requires event handlers
 - ◆ Threads require mutual exclusion in critical sections.

Global control flow

- ❖ Three control flow control mechanisms are used:
 - ◆ procedure-driven control
 - ◆ event-driven control
 - ◆ threads

Procedure-driven control

- ❖ Operations are waiting for input when they need data from actor.
- ❖ Difficult when using object oriented programming languages.
 - ◆ The order of the inputs is difficult to determine from the code because they are scattered between several objects.
- ❖ Example:
 - ◆ `In.readln("Login:");`

Event-driven control

- ❖ The main loop is waiting for an external event.
- ❖ At an external event, the forwarder passes the data to the relevant object.
- ❖ Difficult implementation of procedures with interaction in several successive steps.
- ❖ Example:
 - ◆

```
while (eventStream.hasMoreElements) {  
    subscribers.processEvent(eventStream.nextEvent);  
}
```

Threads

- ❖ The system can create any number of threads, each corresponding to a different event.
- ❖ If the thread requires additional input, it retrieves it (waits) from the specific actor.
sezgisel
- ❖ The most intuitive way to work.
- ❖ Difficult debugging due to the non-deterministic nature of the system.

Borderline cases

sınır vaka(her 2 kategoriye girebilecek durum)

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Handling borderline cases

- ❖ How does the system start up?
- ❖ How the system initializes?
- ❖ How the system shuts down?
- ❖ How should the system react in the case of input data errors?
- ❖ How should the subsystem react in the case of failures of other subsystems?
- ❖ Handling borderline cases affects the interfaces of all subsystems!

Handling borderline cases

❖ Exceptions:

- User error:
Inadequate user input
yetersiz
- HW error:
Network errors, file system errors
- SW error:
The system or subsystem contains an error. An individual subsystem can expect errors from other subsystems and must be protected against them.

Architectural patterns

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

What is a pattern?

- ❖ Christopher Alexander (1977), architecture:
 - ◆ *“Each pattern describes a problem which occurs over and over again in our environment, then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”*
- ❖ Martin Fowler (1997), SW engineering:
 - ◆ *“An idea that has been useful in one practical context and will probably be useful in others”*

What is a pattern?

- ❖ Documented solution of a reoccurring problem.
- ❖ Solution is tested many times and generally accepted.
- ❖ The solution provides a way to solve a problem and not a direct solution that depend on specifics of each individual actual problem being solved.

Architectural patterns

- ❖ The term “pattern” is used in architectures of SW systems:
 - ◆ Architectural patterns or architectural styles.
 - ◆ Each architectural pattern enables designing flexible system architectures that consist of components that are as independent as possible.

Architectural patterns

❖ Patterns to be presented:

- ◆ “Layers”
- ◆ “Model/View/Controller”
- ◆ “Shared repository”
- ◆ “Microkernel”
- ◆ “Client/Server”
- ◆ “Peer-to-peer”
- ◆ “Pipes and filters”
- ◆ ...

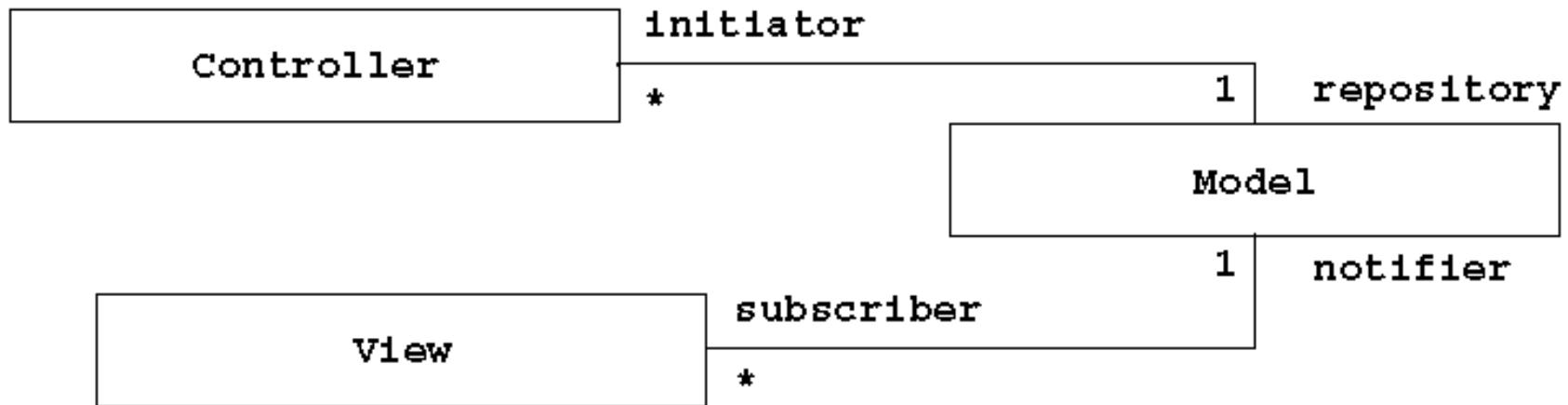
“Layers”

- ❖ Each well designed layer has high cohesion.
- ❖ Coupling between layers is low.
 - ◆ The lower layers do not deal with the specifics of the higher layers.
 - ◆ The only communication is through the API.
 - ◆ Knowledge of the other layers is not required.
 - ◆ Each layer can be designed independently.
 - ◆ The layers can be independently tested.
 - ◆ The lower layers can be designed generically and are therefore reusable.

“Model/View/Controller”

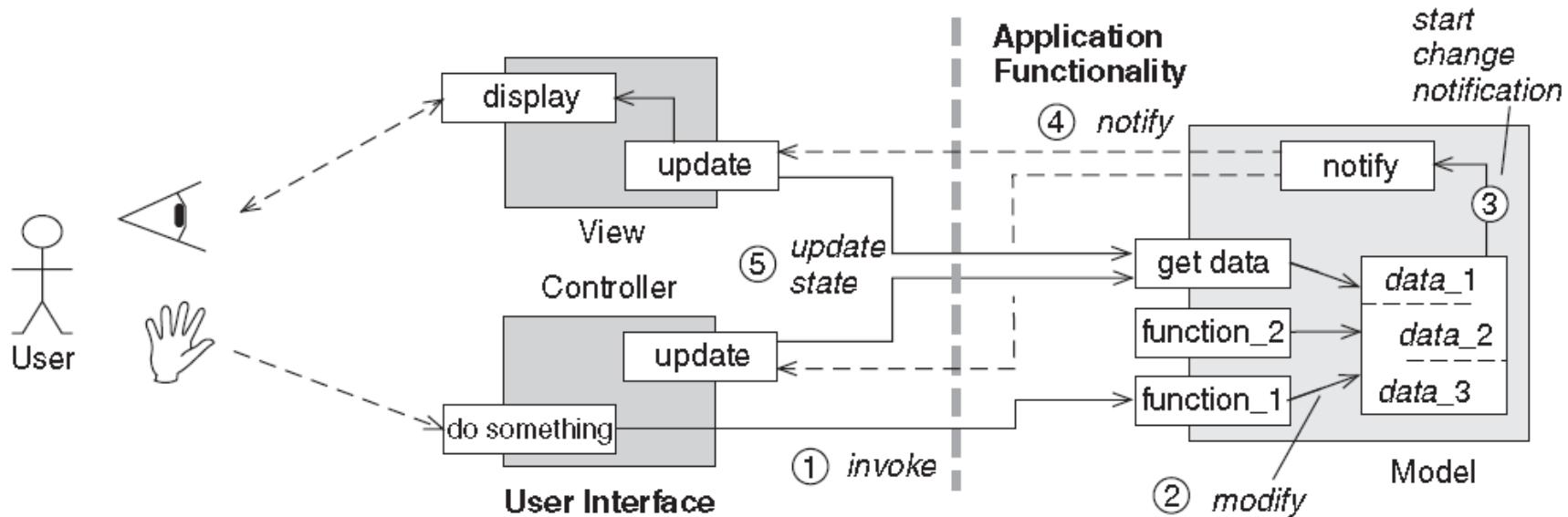
- ❖ The purpose is to separate the user interface layer from other parts of the system:
 - ◆ The model combines all domain knowledge and operations, including objects whose properties we want to display to the user or control through user interaction.
 - ◆ The view provides information to the user.
 - ◆ The controller controls user interaction and transmits user data to the model and view.

“Model/View/Controller”



- ❖ The controller collects user input and sends it to the model in the form of messages. The model maintains a central data structure. The view shows the state of the model. It is notified of any change to the model using the “subscribe / notify” protocol.

“Model/View/Controller”

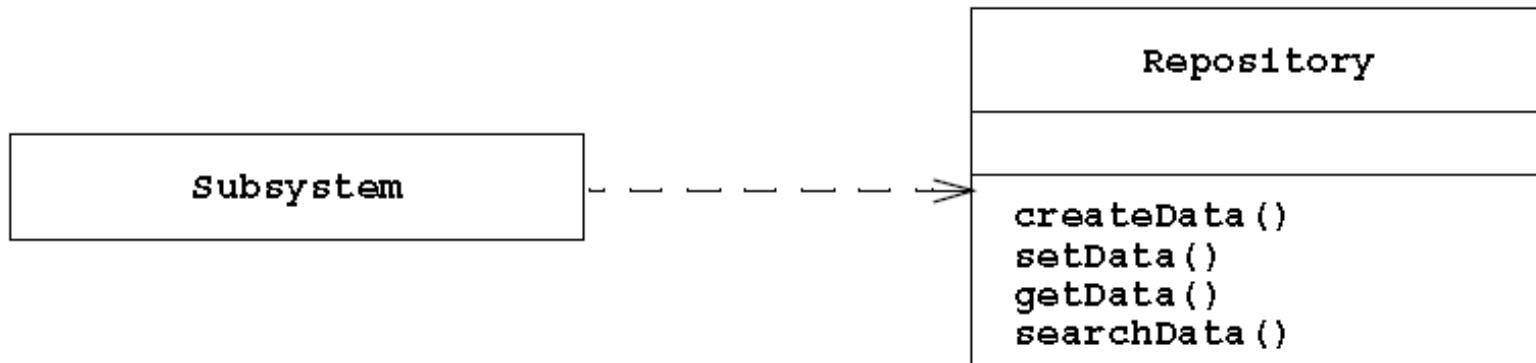


- ❖ The three components can be designed relatively independently.
- ❖ Communication between components is minimal.
- ❖ Ability to test the application (model) without the user interface (view + controller).

“Shared repository”

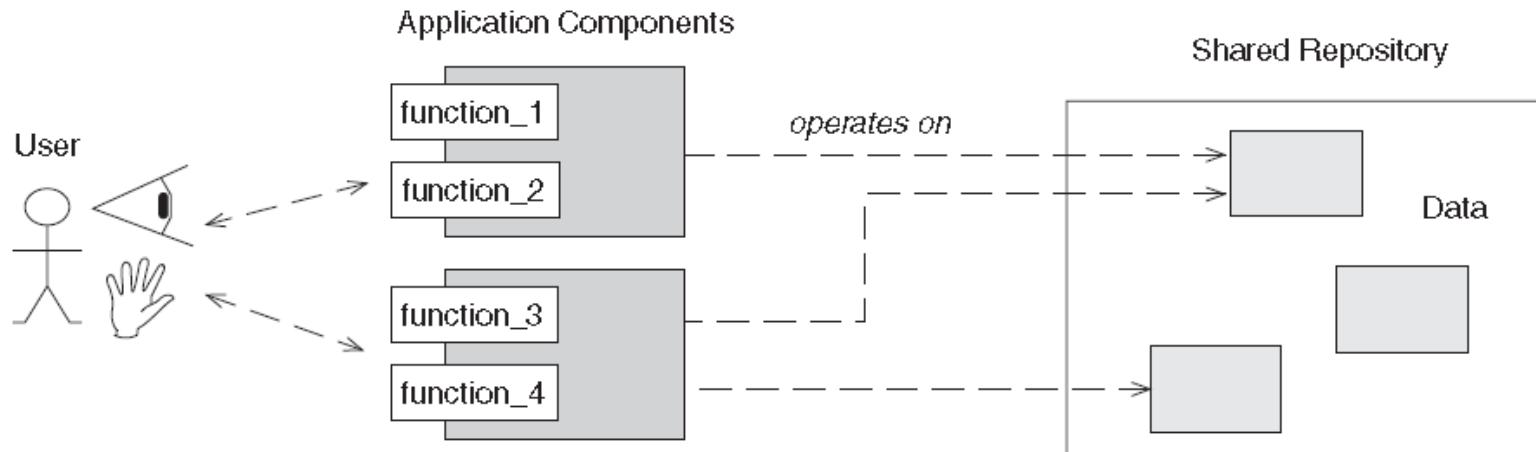
- ❖ Enables central data management:
 - ◆ Multiple subsystems access data from a shared data warehouse.
 - ◆ The subsystems are relatively independent and communicate only through a central data structure.
 - ◆ The control flow can be dictated by the central repository (triggers) or by subsystems (synchronization with the repository lock).
- ❖ Typical architecture of database management systems.

“Shared repository”



- ❖ The operation of each subsystem depends on the data managed by the common repository.

“Shared repository”

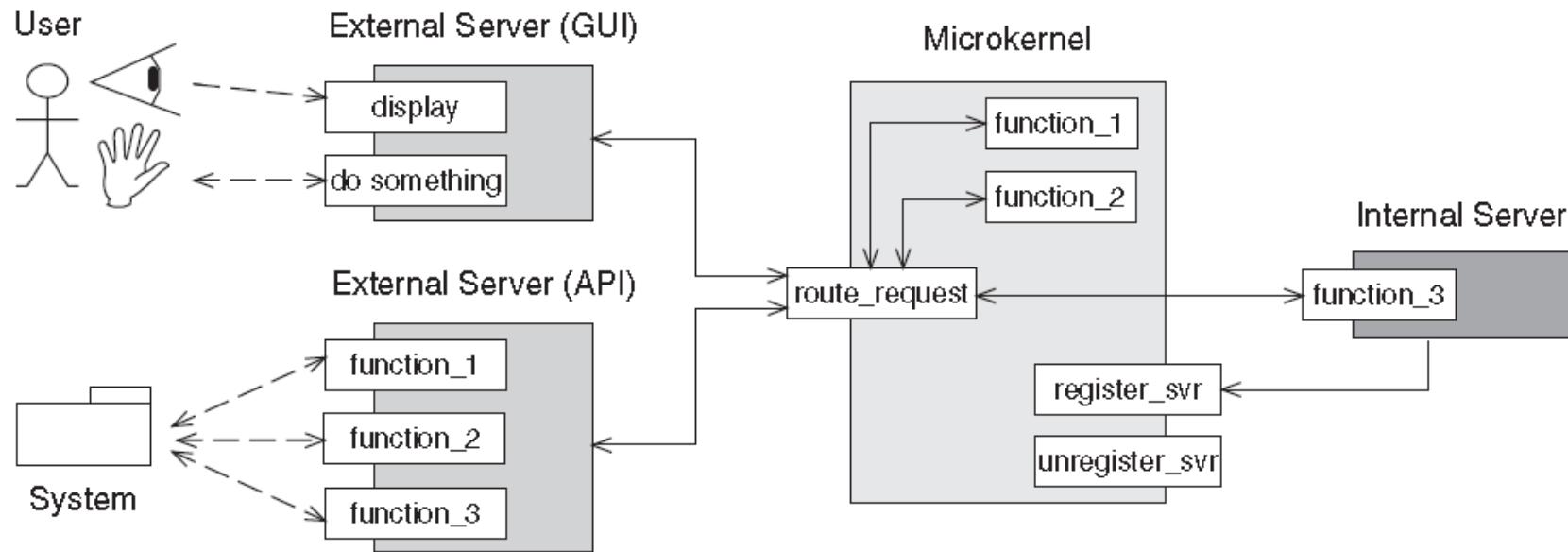


- ❖ All data is stored in a common central repository. All subsystems (application components) are dependent on this repository and access and affect its data.

“Microkernel”

- ❖ Allows different versions of applications to offer different set of functionality or to differ from others by specific properties, e.g. user interface.
- ❖ Despite the differences between the versions, all of them can use the common functional kernel - “Microkernel”.

“Microkernel”

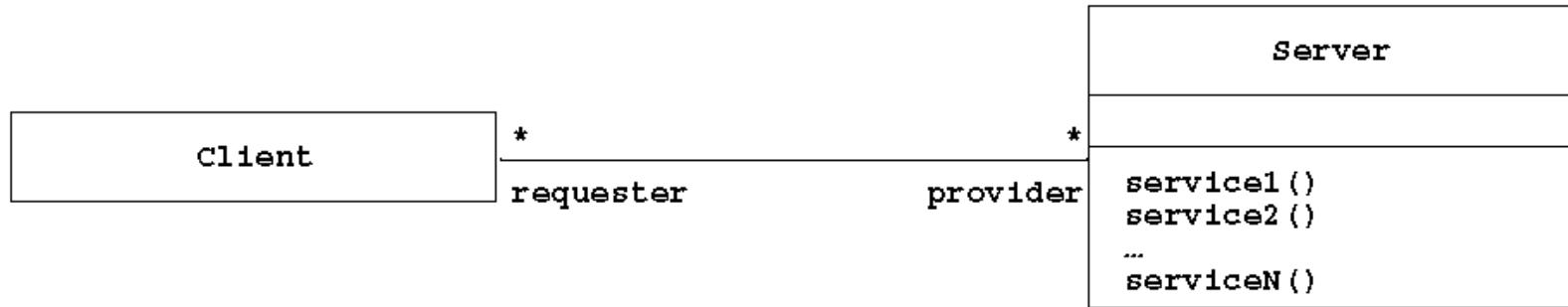


- ❖ Different versions of the application should be built by extending the common function kernel, e.g. using plug-and-play infrastructure.

“Client/Server”

- ❖ A distributed system in which the server subsystem provides services to the client subsystems.
- ❖ The client request is usually sent via a remote procedure call mechanism or a common object broker, such as CORBA or Java RMI.
- ❖ The control flow between clients and servers is independent, with the exception of query / result synchronization on servers.

“Client/Server”

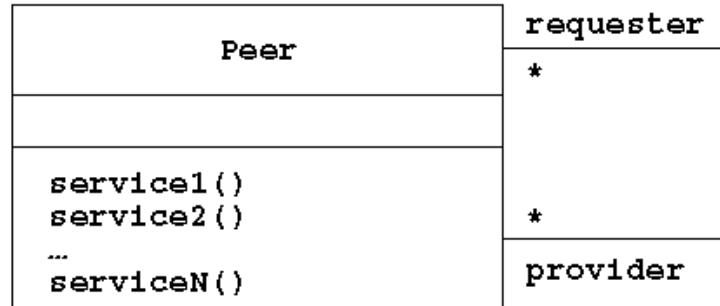


- ❖ Clients query the services of one or more servers.
- ❖ Servers do not know clients.
- ❖ The "client / server" architecture represents a generalization of the "Shared repository" architecture.

“Peer-to-peer”

- ❖ Architecture is a generalization of the “Client / server” architecture in which each subsystem can act as a server and as a client.
- ❖ Each subsystem accesses the services of others and offers its own services.
- ❖ The control flow of the subsystems is independent, with the exception of accesses synchronization.

“Peer-to-peer”



- ❖ Designing peer-to-peer systems is more difficult than client / server systems because of the deadlock threat.

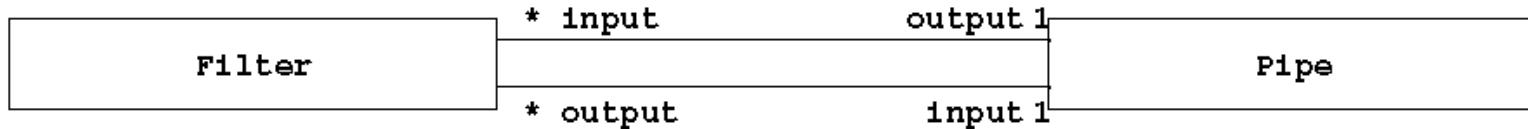
“Pipes and filters”

- ❖ Subsystems process the data, which they obtain from other subsystems through a series of inputs, and pass it on to other subsystems via a series of outputs.
- ❖ The subsystems are called filters, the associations between them are called pipes.
- ❖ The architecture is variable - it is possible to change filters or change the configuration to achieve other goals.

“Pipes and filters”

- ❖ The relatively simple format of the data stream is transmitted through a series of filter subprocesses:
 - ◆ Each filter processes the data in its own way
 - ◆ The architecture is very flexible:
 - Almost any filter can be omitted.
 - It is possible to replace the filter.
 - New filters may be added.
 - It is possible to change the order of the filters.
 - ...

“Pipes and filters”

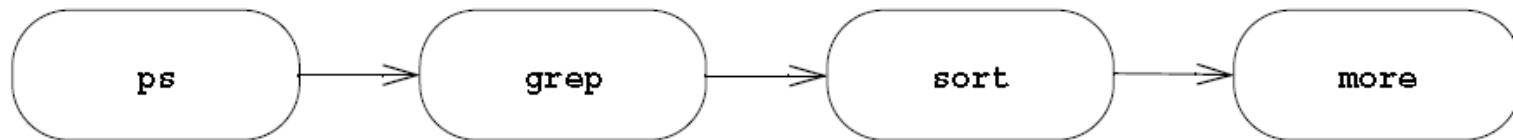


- ❖ Suitable architecture for systems requiring data flow transformation without user intervention.
- ❖ Not suitable for systems with more complex component interactions.

“Pipes and filters” – example 1

```
% ps auxwww | grep dutoit | sort | more

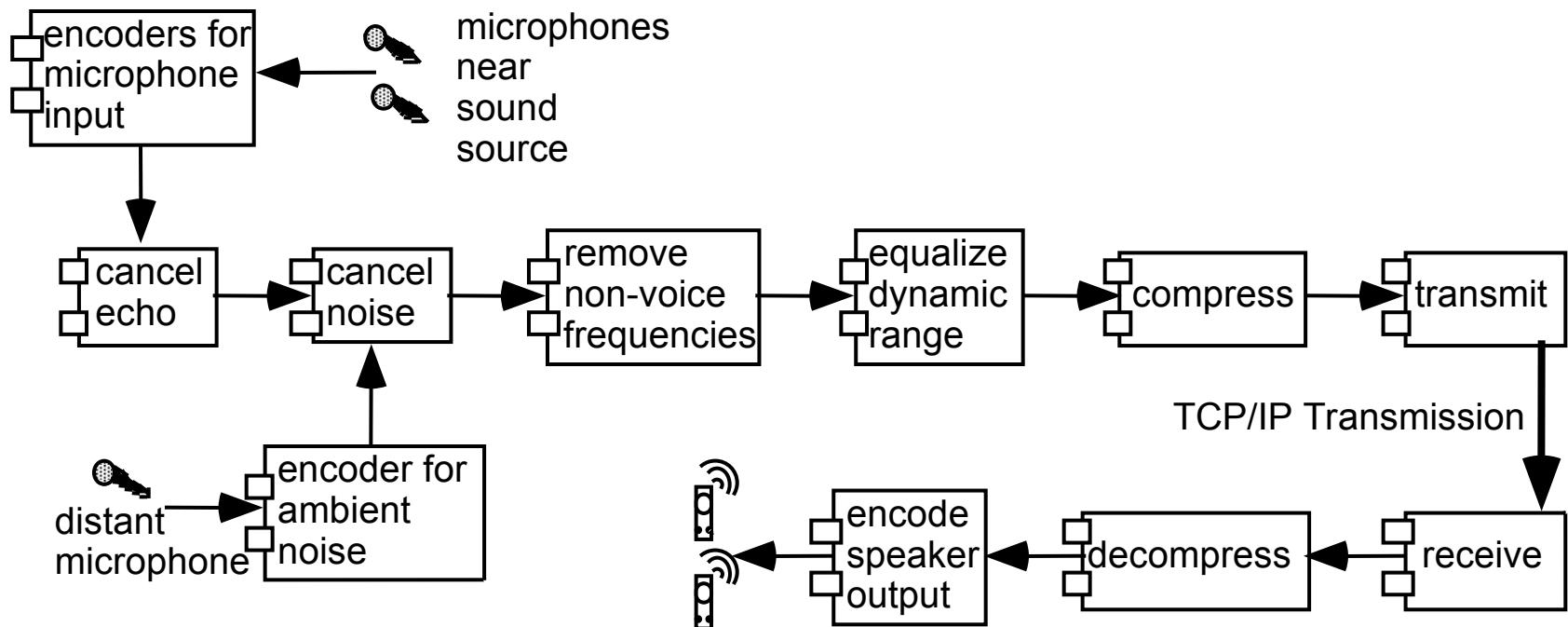
dutoit  19737  0.2  1.6 1908 1500 pts/6      0 15:24:36  0:00 -tcsh
dutoit  19858  0.2  0.7   816   580 pts/6      S 15:38:46  0:00 grep dutoit
dutoit  19859  0.2  0.6   812   540 pts/6      0 15:38:47  0:00 sort
```



❖ Unix shell:

- ◆ ps (process status)
- ◆ grep (search for pattern) – here used to find the processes of a particular user
- ◆ sort (arranging the order)
- ◆ more (display at the terminal, one screen of text at a time)

“Pipes and filters” – example 2



“Pipes and filters”

- ❖ The filters can be designed independently and have functional cohesion.
- ❖ The same filters can be used multiple times for multiple applications.
- ❖ Possible independent testing of individual filters.

Architectural patterns in general

- ❖ There are various architectural patterns, not only those presented here.
- ❖ There are connections (similarities, generalizations) between architectural patterns
- ❖ Architectural patterns can be combined - multiple patterns are used simultaneously to determine the architectural solution to a problem.

Documenting

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

An architectural model contents

- ❖ A good architectural model presents the system from multiple "views":
 - ◆ logical division into subsystems
 - ◆ interfaces between subsystems
 - ◆ dynamics of interaction between components during run time.
 - ◆ common subsystem data
 - ◆ deployment of components / subsystems to devices (HW).

Architectural model development

- ❖ Modeling begins with sketching an outline of the architecture
 - ◆ Depending on basic requirements and use cases.
 - ◆ Determination of basic system components.
 - ◆ Selection of architectural patterns used.
 - ◆ It is useful for several independent groups to prepare the initial drafts separately and then combine them and apply the best ideas.

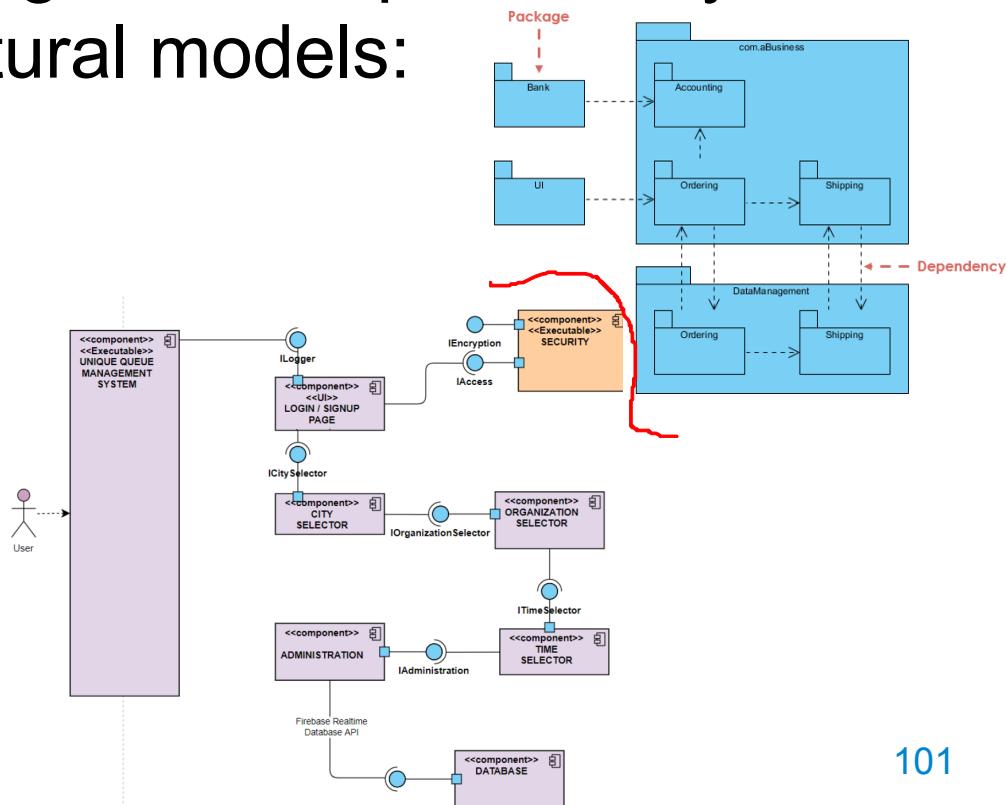
Architectural model development

❖ Improving architecture

- ◆ Identification of component interactions and determination of associated interfaces.
- ◆ Distribution of data and functionality between system components.
- ◆ Identifying options for reusing existing frameworks or deciding to build a new framework.
- ◆ Consideration of each use case and customization of the architecture for its implementation.

UML description of an architecture

- ❖ All UML diagrams can be useful for describing different aspects of an architectural model.
- ❖ The following UML diagrams are particularly important for architectural models:
 - ◆ package diagrams
 - ◆ component diagrams
 - ◆ deployment diagrams



Software Architecture Plan

- ❖ Documenting architecture contributes to better architectural solutions:
 - ◆ Documentation forces important decisions to be made before implementation begins.
 - ◆ Enables reviews of the architecture and thus offers the potential for further improvements.
 - ◆ Facilitates / enables / presents communication
 - with those who will build the system,
 - with those who will change the architecture of the system in the future,
 - With those who will build other systems that will connect to the planned one.

Architectural plan structure

- ❖ **A. Purpose:**
 - ◆ Which system or subsystem the architectural design refers to.
 - ◆ Reference to requirements that are met (realized) by the planned system.
- ❖ **B. Priorities considered**
 - ◆ A description of the design goals and priorities that were used in the design of the system.
- ❖ **C. Summary of the Architectural Plan:**
 - ◆ A general high-level description of the architectural plan, with which the user becomes familiar with the proposed solution.
- ❖ **D. Description of the proposed architecture:**
 - ◆ Identify the main challenges that need to be addressed.
 - ◆ Description of alternatives and decisions on the chosen alternative, explaining the choice.
- ❖ **E. Other details of the architectural plan:**
 - ◆ Details that have not yet been mentioned and are relevant to the reader.

Guidelines

- ❖ Avoid listing information that is obvious to an experienced planner or programmer.
 - ◆ details that would be more appropriate for commenting on program code.
 - ◆ Implementation details of individual subsystems that are not relevant to understanding the system as a whole.
 - ◆ ... details ...

Architectural plan recension

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Recension

Review and evaluation of an architectural plan by a qualified individual or group ...

Recension

- ❖ In contrast to the recension of the SW requirements specification, the recension of the architectural plan does not include external reviewers (clients, users ...).
- ❖ The review should be done by developers who were not involved in the design.
- ❖ Reviewers should have sufficient interest in the in-depth study of the proposed solution.

Architectural model correctness

- ❖ An architectural model is correct if it permits the mapping of the requirements specification model to the system model:
 - Can each subsystem be linked to use cases or non-functional requirements?
 - Can each use case be linked to a group of subsystems?
 - Can design goal be linked to non-functional requirements?
 - Are all non-functional requirements met?
 - Is there an access control method for each actor?

Architectural model completeness

- ❖ Does the model meet all the requirements?
 - ◆ Were borderline cases taken into account?
 - ◆ Does the model meet all defined use cases?
 - ◆ Are all aspects of architectural planning addressed: HW / SW connections, data storage, access control, borderline cases)
 - ◆ Are all subsystems defined.

Architectural model consistency

- ❖ Are there any contradictions?
 - Are conflicting design goals prioritized?
 - Do any of the design goals conflict with the non-functional requirements?
 - Do multiple subsystems or classes have the same name?

Architectural model reality

- ❖ Can the system be developed?
 - ◆ Are new technologies or new components included? Are they appropriate and robust enough for the designed system?
 - ◆ Have performance and reliability requirements been analyzed?
 - ◆ Have concurrency needs been taken into account?

Architectural model readability

- ❖ Can the model be understood, even for people who have not been involved in modeling?
 - ◆ Are the names of the subsystems clear?
 - ◆ Do objects (subsystems, classes, operations) with similar names describe similar phenomena?
 - ◆ Are all objects described with the same level of details?

Software engineering

Component design

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

When to design components?

❖ **System analysis**

- ◆ Decomposes the system into components that are easier to describe (fulfilling the requirements).

❖ **System design**

- ◆ System decomposition, selection of a global control flow mechanism, HW/SW relations, persistent data management, access control method...
- ◆ Integrating existing reusable components.

❖ **Component design (objects)**

- ◆ Planing of the reuse of the existing components (libraries, wrapping, design patterns).
- ◆ Filling in the gaps between application objects and selected existing components.
 - Identification of additional objects
 - Improvements of exisiting objects (understandability, scalability, extensibility)
 - Detailed interface specification (e.g. for all classes)
 - Optimization of object design (Responsiveness, Memory ...)

OO and structural approach

- ❖ Object oriented aproach (OOAD):
 - ◆ System design
 - Decomposition into subsystems
 - ◆ Component/object design
 - Selection of implementation programming language.
 - Selection of algorithms, definition of data structures.
- ❖ Structural analysis and design (SAD):
 - ◆ Preliminary Design
 - Decomposition into subsystems
 - Definition of data structures.
 - ◆ Detailed Design
 - Selection of algorithms.
 - Finishing definition of data structures.
 - Selection of implementation programming language
 - Usually joined with preliminary design (a single phase).

Component design

- ❖ We will focus on object oriented approach of object design.
- ❖ Component design is not algorithmic!
 - ◆ The result is influenced by skills (knowledge, experience, creativity...)
- ❖ Object design steps:
 - ◆ Using existing components
 - ◆ Specifying interfaces
 - ◆ System restructuring
 - ◆ Optimization

Component selection

- ❖ Selection of existing solutions:
 - ◆ off-the-shelf class libraries
 - ◆ frameworks
 - ◆ components
- ❖ Customization of existing libraries, frameworks and components
 - ◆ Changing the API when source code is available
 - ◆ "Adapter" or "bridge" design pattern if source code is not accessible.
- ❖ Designing new components based on the specified software architecture.

Communication between objects

- ❖ Conceptually, objects communicate by passing messages.
- ❖ The messages consist of:
 - ◆ The names of the service called by the caller object from the called object.
 - ◆ Information needed to perform the service.
- ❖ In practice, messages are often implemented in the form of procedure calls.
 - ◆ Name = procedure name;
 - ◆ Information = parameters.

Message examples

```
// Call a method associated with a buffer  
// object that returns the next value  
// in the buffer
```

```
v = circularBuffer.Get () ;
```

```
// Call the method associated with a  
// thermostat object that sets the  
// temperature to be maintained
```

```
thermostat.setTemp (20) ;
```

Object design guidelines

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

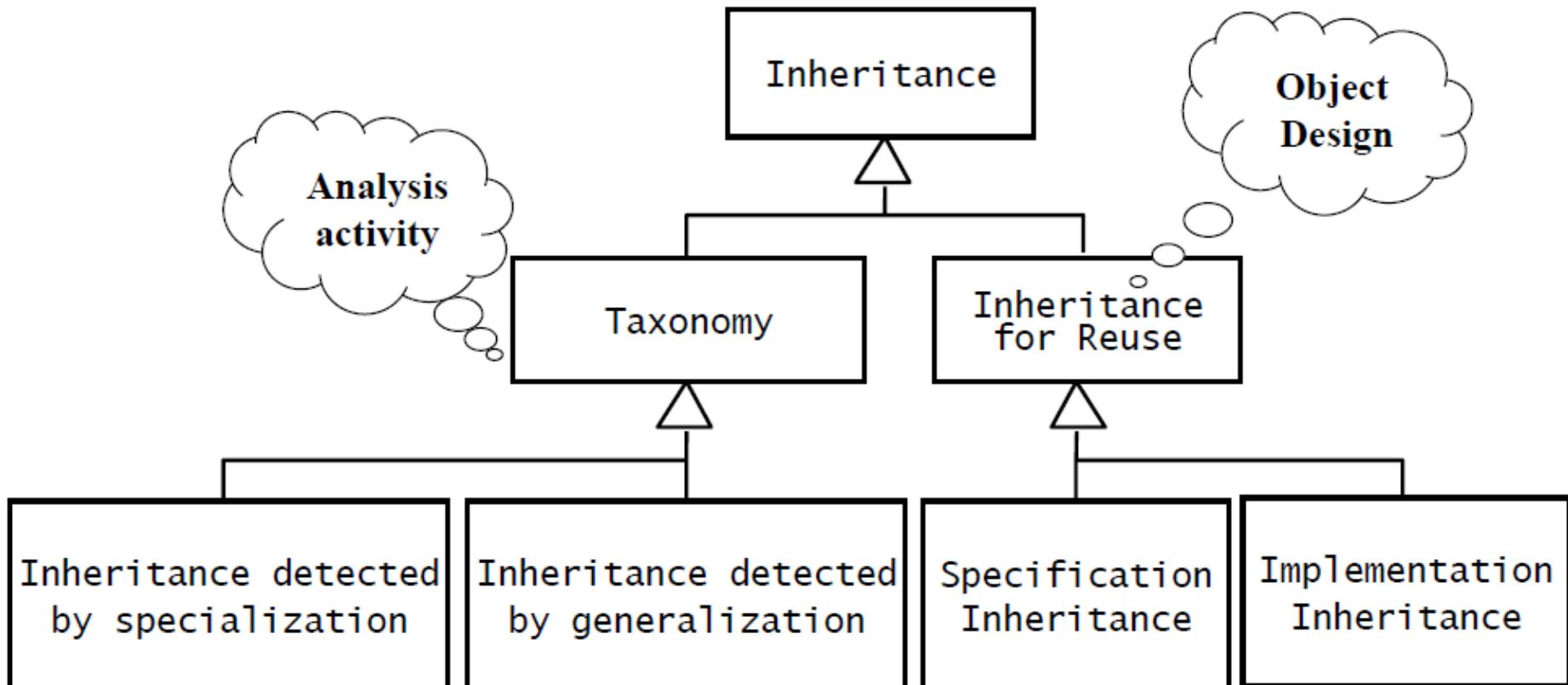
Design guidelines

- ❖ Object-oriented planning uses different approaches, including:
 - ◆ the use of inheritance,
 - ◆ delegation,
 - ◆ abstraction / wrapping,
calling other func.
 - ◆ modularity.

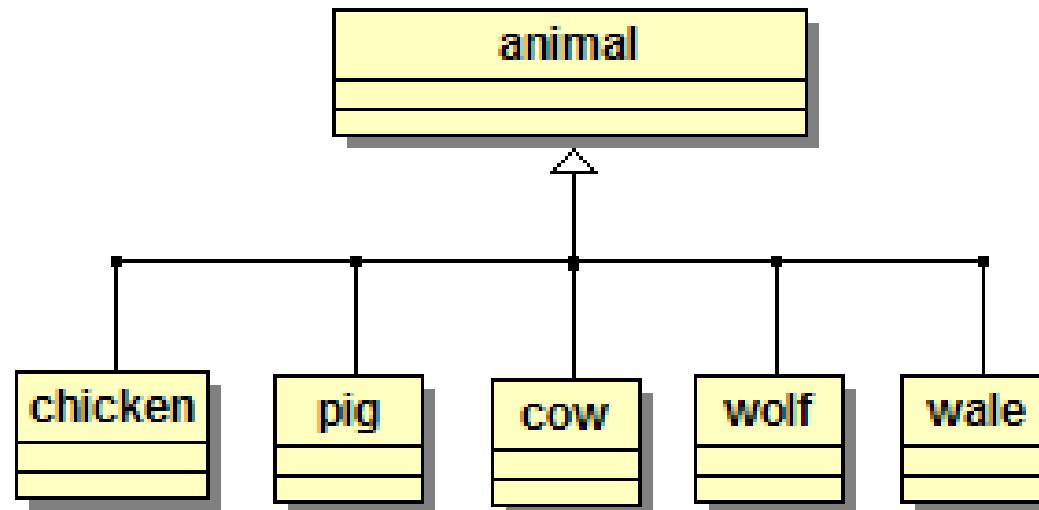
Inheritance

- ❖ Inheritance is used to achieve two different goals:
 - ◆ Ordering of classes (taxonomy)
 - It is used in the requirements analysis phase
 - to identify objects that are hierarchically interdependent.
 - It helps to understand the models
 - ◆ Reuse
 - It is used in the design phase.
 - It contributes to the ability to use existing objects and to modify and extend them
 - Extension of functionality (implementation inheritance) or Interface specification (specification inheritance)

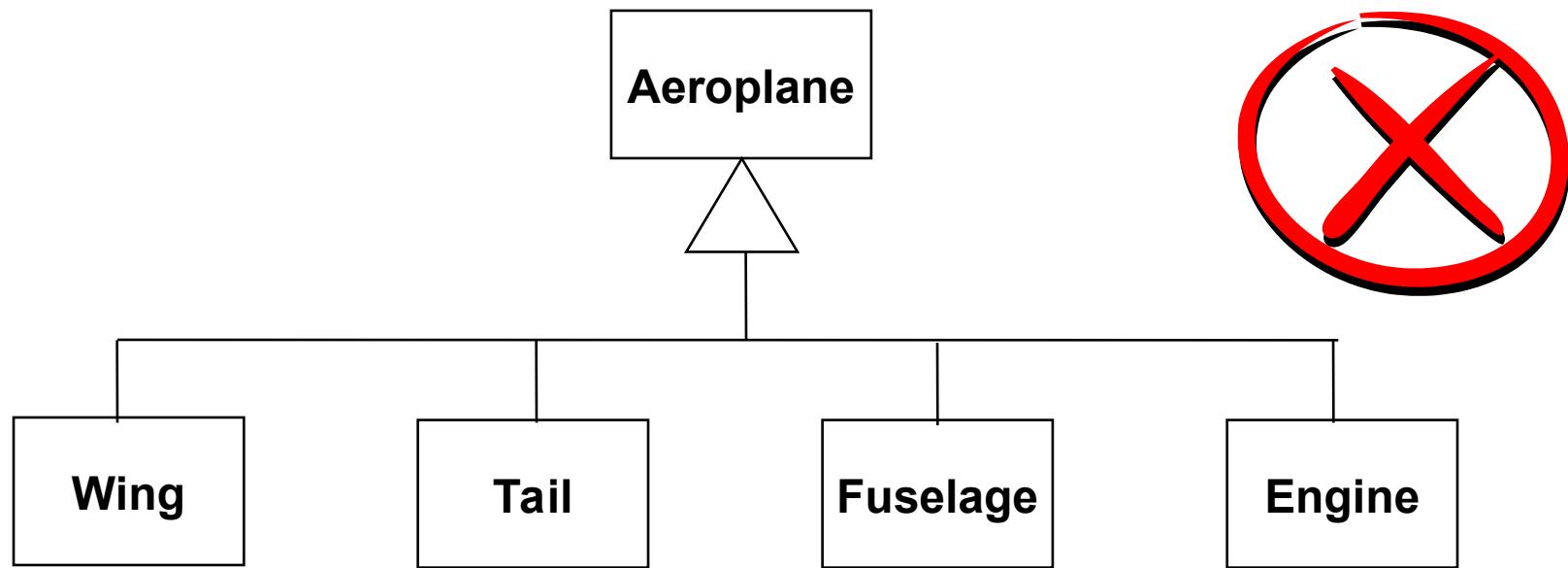
Inheritance metamodel



Inheritance for taxonomy

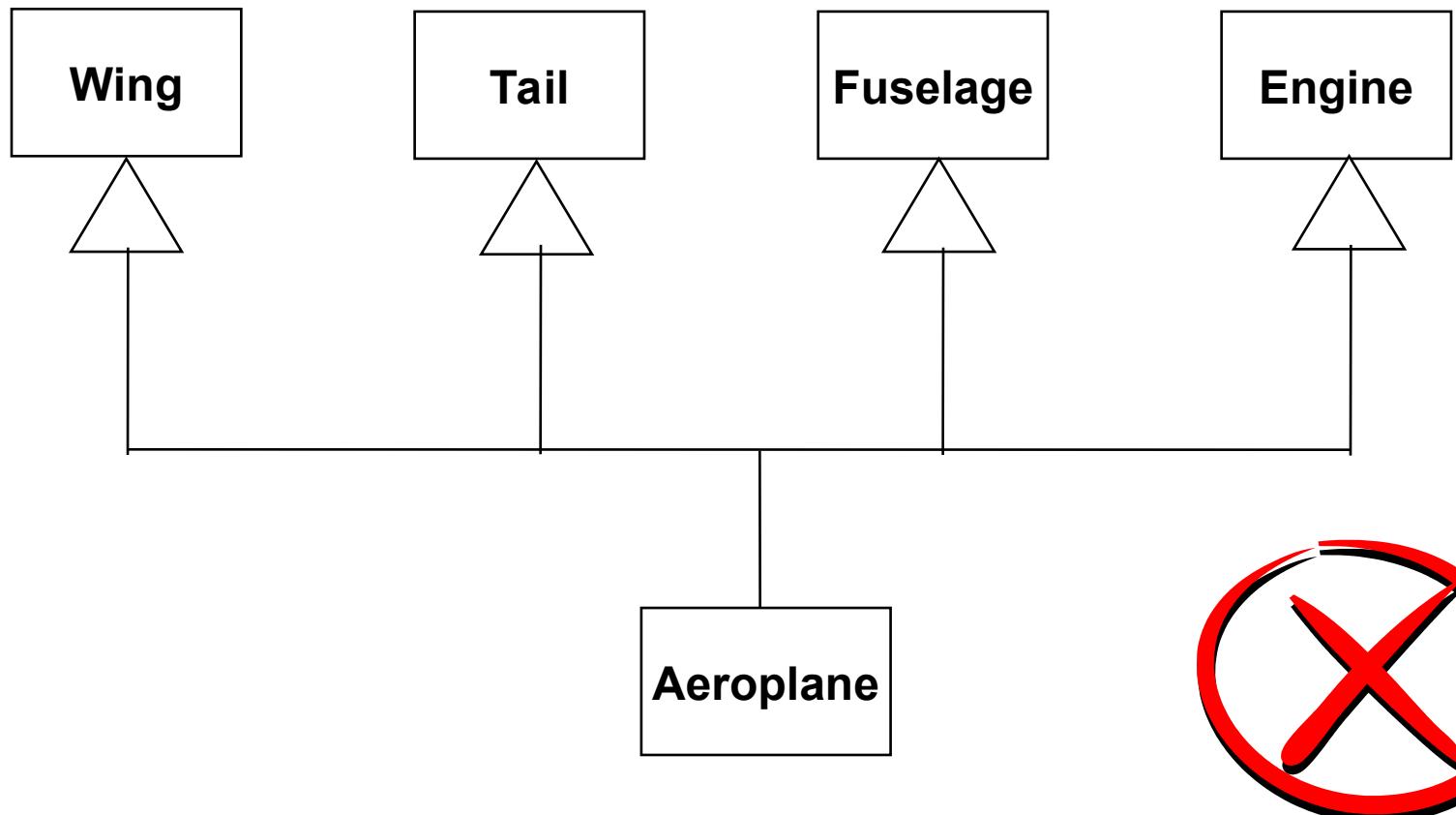


Inheritance for taxonomy?

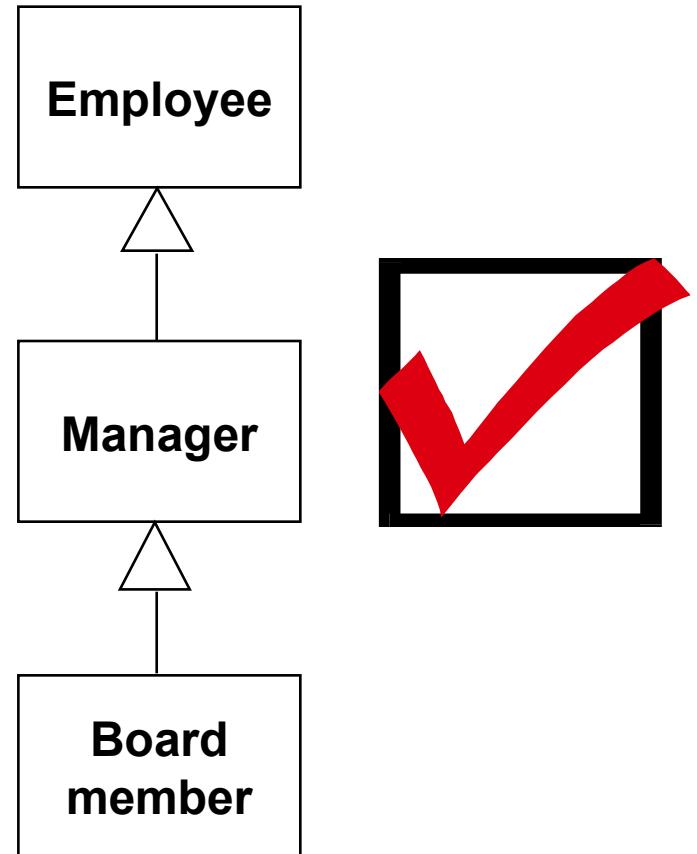
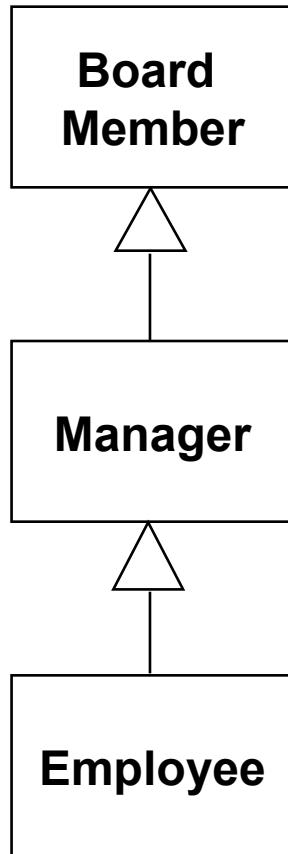


- ❖ Misuse – inheritance instead of aggregation.

Inheritance for taxonomy?

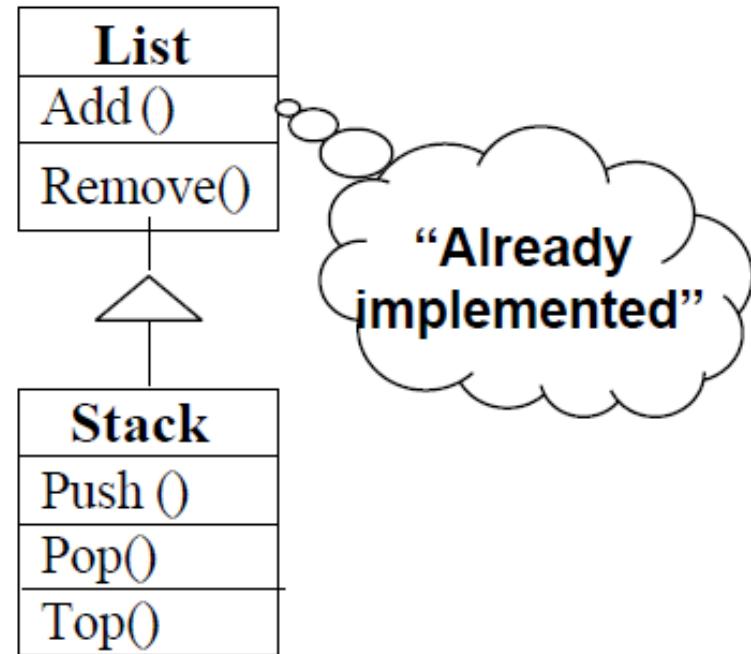


Inverted inheritance hierarchy



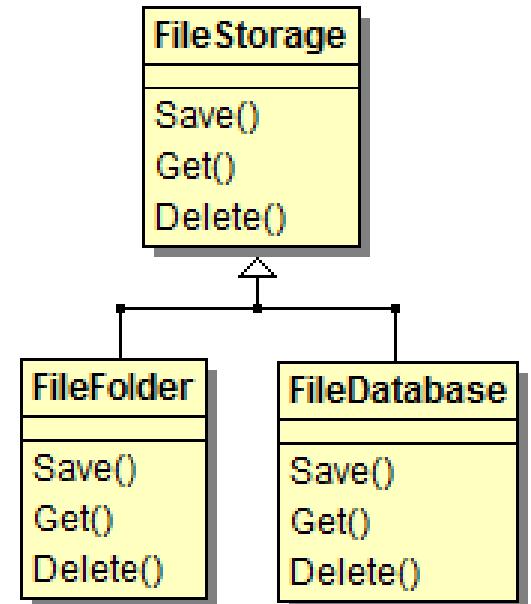
Implementation inheritance

- ❖ Using of similar class with similar functionality
- ❖ Example:
 - ◆ An existing class **List** can be used to build a new class **Stack**.
 - ◆ New methods **Push()**, **Pop()**, **Top()** will be built on existing functionality of methods **Add()** and **Remove()**.
- ❖ Attention: methods of the basic class are available also available in the new class that inherit from the basic one.



Specification inheritance

- ❖ Inherited interface.
- ❖ The subclass implement methods (its own way), defined by a parent class that defines the interface.
- ❖ Enables different implementations with the same interface – external functionality.

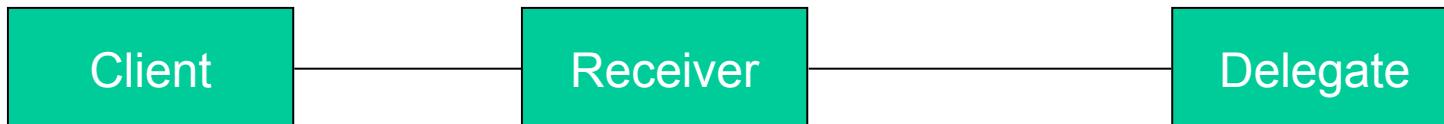


Inheritance for reuse

- ❖ Implementation inheritance)
 - ◆ The goal is to extend functionality while using the existing one.
- ❖ Specification or Interface inheritance
 - ◆ The goal is to define an interface for using different implementations of required functionality.

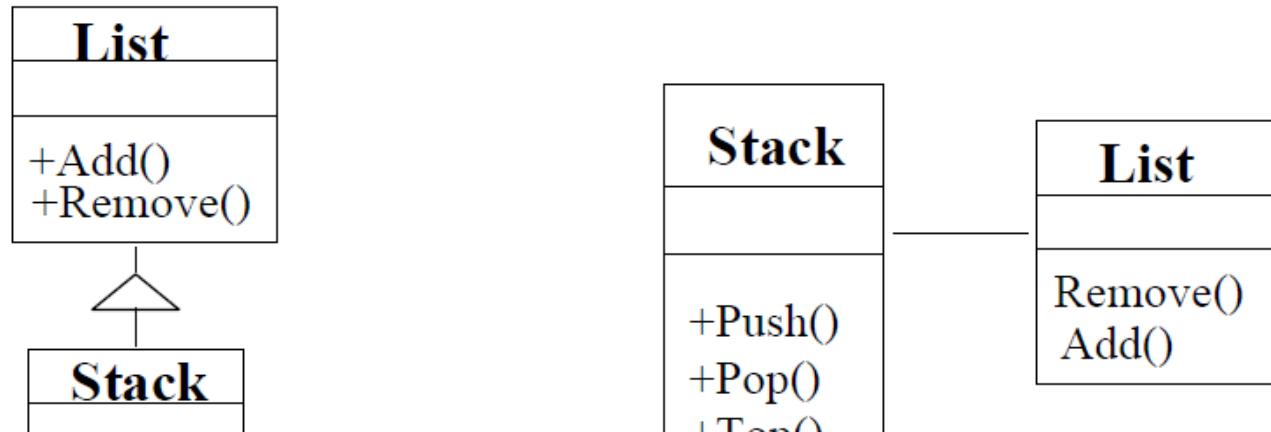
Delegation

- ❖ Delegation is a way towards component reuse.
- ❖ A request from the client is received by a receiver that passes it (modified) to the delegate.
 - ◆ The receiver makes sure that the delegate is properly used preventing inappropriate usage.



Delegation or inheritance?

❖ Example:



Delegation and inheritance

❖ Delegation

- ◆ + Flexibility
- ◆ + Possible in all programming languages
- ◆ - Reduced performance due to encapsulation

❖ Inheritance

- ◆ + Simplicity
- ◆ + Supported in many programming languages
- ◆ + Easily to add new functionality
- ◆ - Exposure of parent class details
- ◆ - Any change in the parent class requires a change in the inherited class

Minimization of dependency

- ❖ SW elements are dependent whenever a change of one SW element requires the change of some other SW element.
- ❖ Dependencies are the main source of difficulties during maintenance.
- ❖ SW design tends towards minimized dependencies.
- ❖ Measure of software dependencies is called Connascence

Examples of Connascence

❖ Dynamic connascence:

- ◆ Connascence of Execution (CoE) – e.g., required initialization before use
- ◆ Connascence of Timing (CoT) – e.g. the irradiation device shutdown command must follow the device activation command in less than 50 milliseconds.
- ◆ Connascence of Values (CoV) - e.g. arithmetic limit - the sum of all angles of a triangle is always equal to 180 degrees.
- ◆ Connascence of Identity (CoI) - e.g. if two objects (O1 and O2) have to refer to the same O3 object (eg with pointers)
- ◆ Diversity - e.g. if class C inherits from classes A and B, then A and B must not contain methods of the same name.

❖ Static...

- ◆ Connascence of Name, Type, Meaning, Position, Algorithm.

Liskov substitution principle

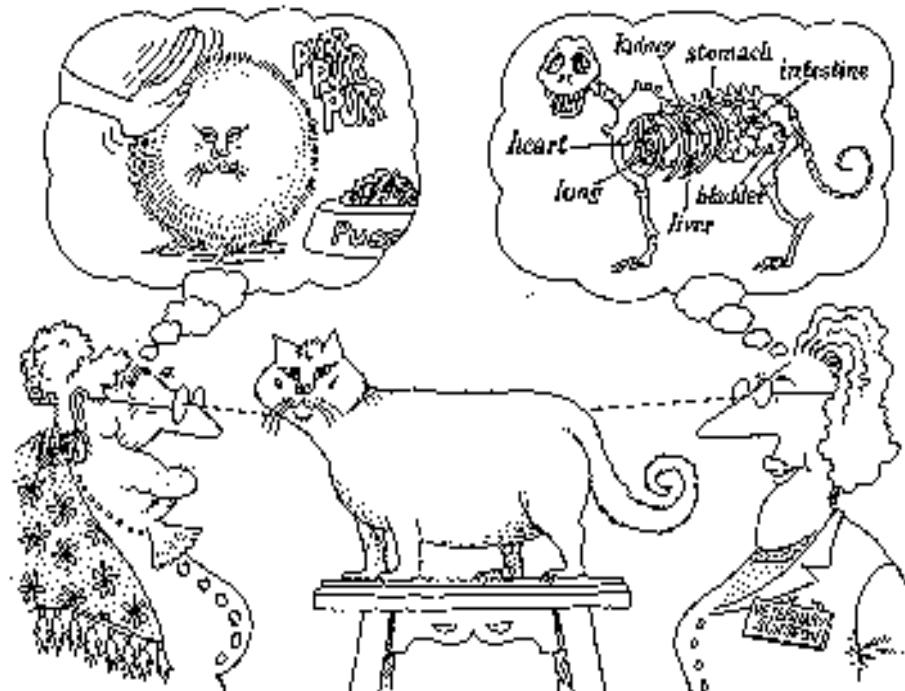
- ❖ After Barbara Liskov and her lecture “*Data abstraction and hierarchy*” 1987.
- ❖ if class S **is a subclass** of class T, then objects of class T may be replaced with objects of class S without altering any of the desirable properties of the program (correctness, task performed, etc.)
- ❖ If the base class meets the requirements of the application, then the LSP replacement class must also meet the same requirements so that from the interface point of view it works exactly the same and produces the same results for the same arguments.

LSP examples

- ❖ 2D game board
- ❖ Extension to 3D
- ❖ All existing features contain only X and Y arguments, not Z !!!
- ❖ Violation of LSP !!!
- ❖ The solution ... ???
- ❖ Class "rectangle"
- ❖ Replaced with "square" class.
- ❖ !!! The length of the sides is no longer independent !!!
- ❖ Violated LSP !!!

Abstraction

- ❖ Indicates the essential properties of an object that sets it apart from other objects.
- ❖ It defines the conceptual boundaries of objects, while allowing different interpretations depending on the view the observer



Example – Light switch ON/OFF

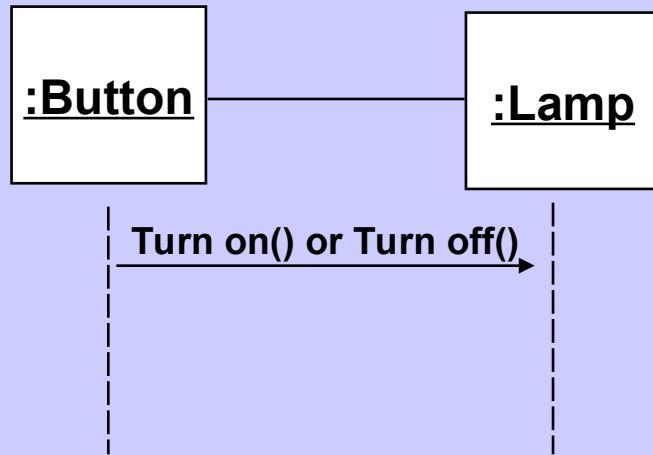
Use case:

Name: Turn lamp on and off

Actors: ButtonPusher

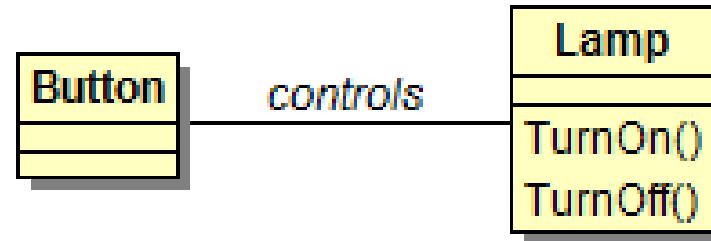
Description: When the ButtonPusher presses the button, the lamp goes on if it was off, and goes off if it was already on.

Sequence diagram:

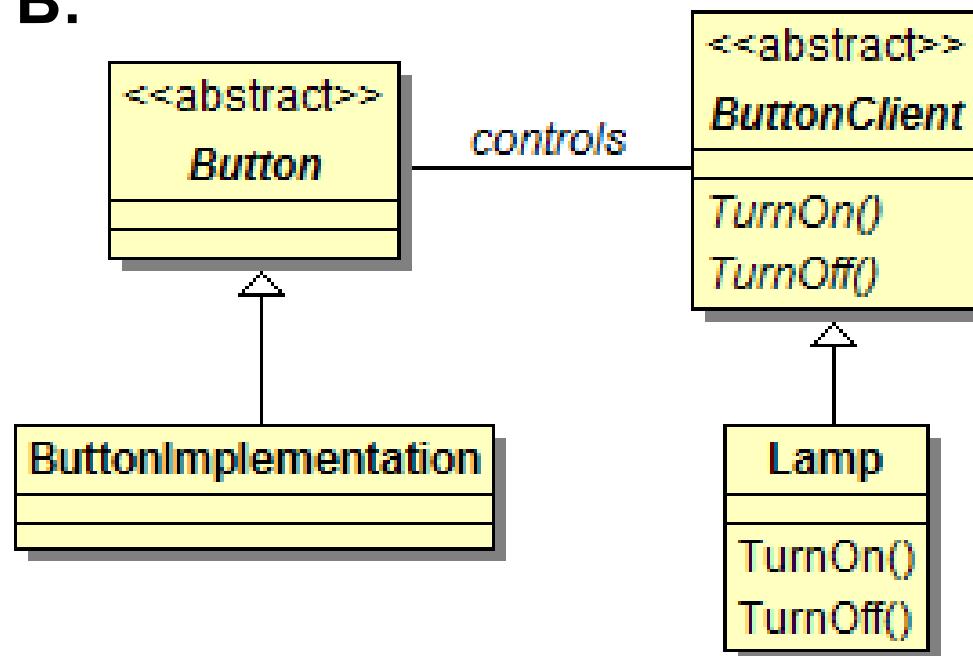


Example – Light switch ON/OFF

Solution A:



Solution B:



Example – Light switch ON/OFF

❖ Abstraction:

- “On / off” relation between the button and the target object
- Implementation is separate from abstraction

❖ Why is solution B better?

- ◆ Button implementation, e.g. the way the user interacts is irrelevant to the interaction of the button with the light.
- ◆ The button does not require knowledge of the lamp.
- ◆ The solution makes it easier to introduce changes.
- ◆ The solution can be reused for other purposes.
- ◆ The requirements are still easily traceable

The open/closed principle

- ❖ Keep the class open for extensions - adding operations or data structure fields
- ❖ The class should be closed for changes if available for use by other classes - it must specify a stable and well-defined interface.
- ❖ In object-oriented programming languages, this is realized through inheritance and polymorphism.

Encapsulation

- ❖ Encapsulation is the process of wrapping elements by defining an abstraction that determines the structure and behavior of the elements.
- ❖ Encapsulation is used to separate "contract" interfaces (defined by abstraction) from implementation.

Class cohesion

- ❖ Class cohesion measures the interconnectedness of the methods of the class's external interface.
- ❖ How the class works in terms of implementing abstraction.
- ❖ We want high cohesion!
- ❖ There is no quantitative measure of class cohesion.
- ❖ Symptoms of low cohesion:
 - ◆ A class contains some components (methods) that are not defined for all class objects. Possible improvement with separation into two classes.
 - ◆ The class contains components (methods) that are not relevant to the purpose of the class (domain).

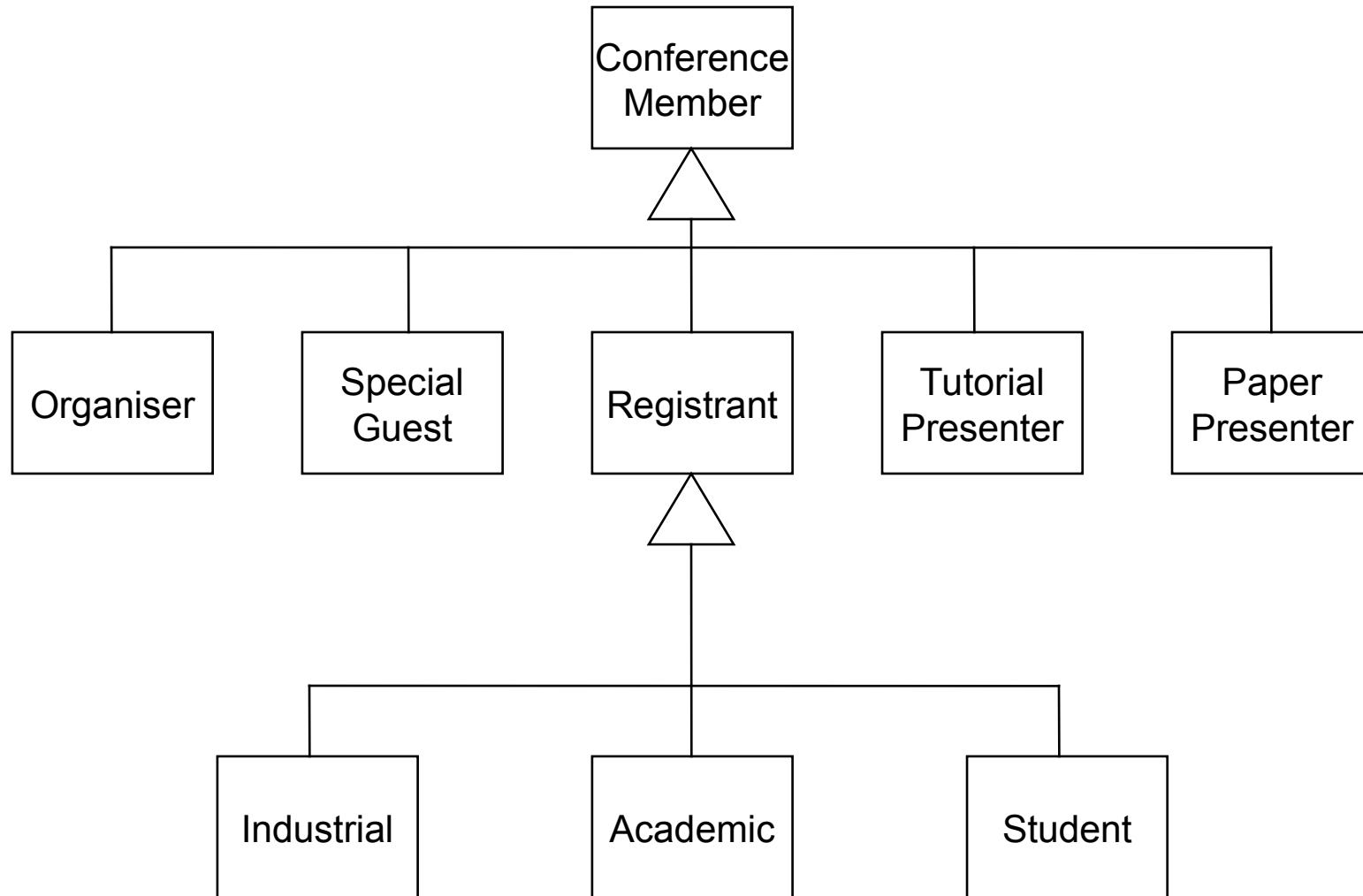
Use of "roles" instead of inheritance

- ❖ Inheritance can often be replaced by the use of roles.
- ❖ Example: Scientific Conference Registration System:
 - ◆ Participants can be defined by the following groups:

Conference members:

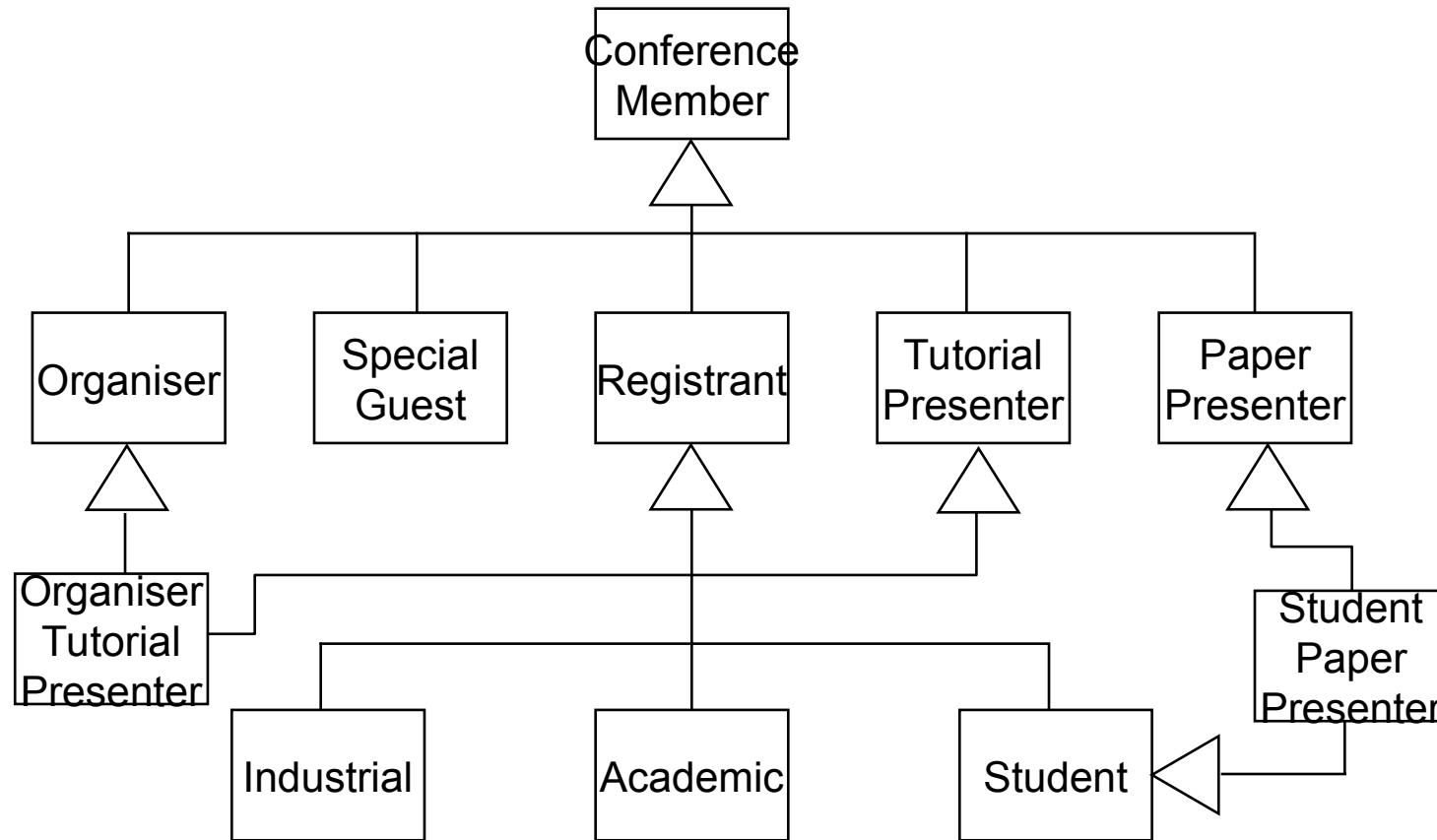
- Organisers
- Special Guests
- Tutorial Presenters
- Paper Presenters
- Industrial Registrants
- Academic Registrants
- Student Registrants

Solution with inheritance



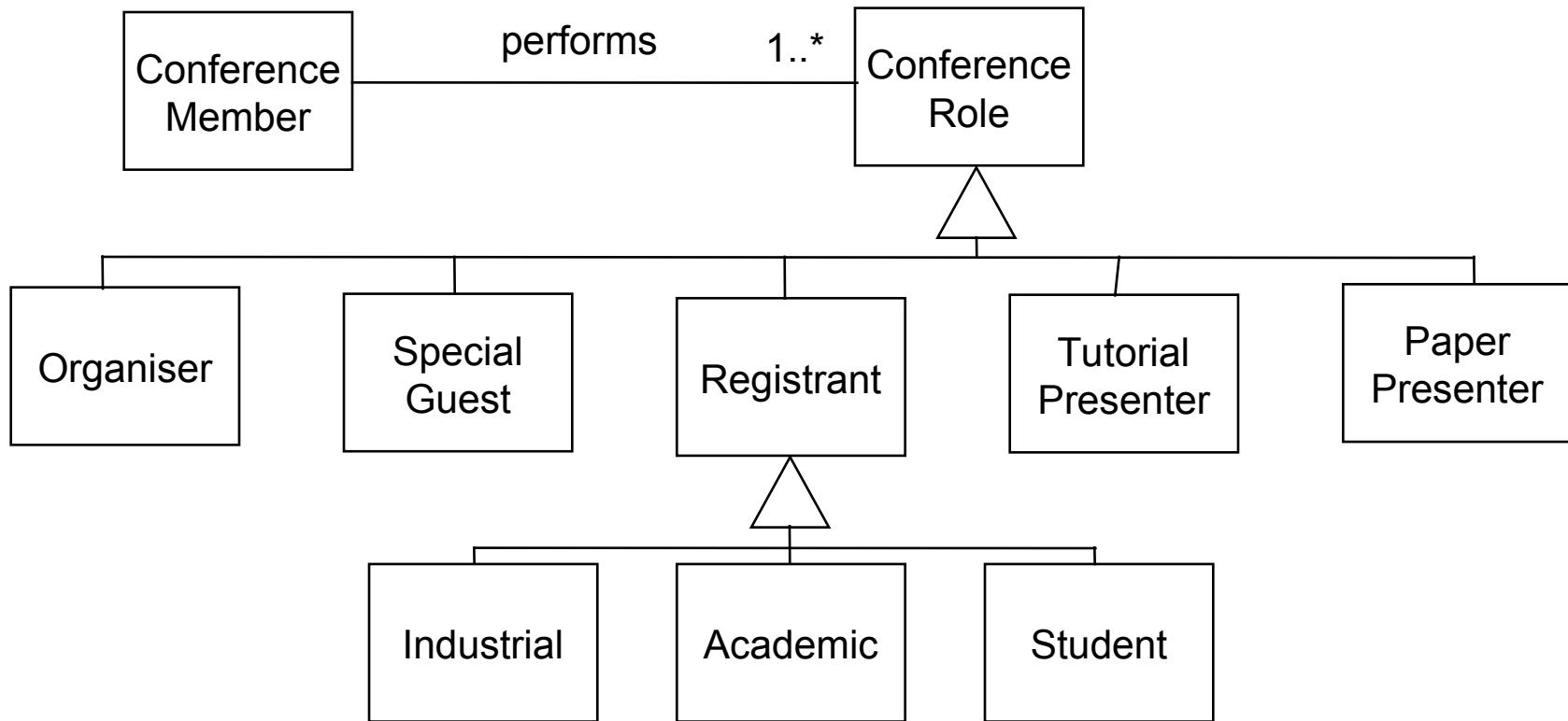
Inheritance problems

- ❖ The organizer can also give a lecture!
- ❖ A student can also present a paper!
- ❖ ...



A solution using roles

- ❖ Roles provide a flexible way of avoiding multiple inheritance.



General – modularity!

- ❖ The modular program consists of well-defined, conceptually simple and independent units - modules that communicate through well-defined interfaces.
- ❖ Advantages of modularity
 - ◆ easier to understand and interpret,
 - ◆ easier documentation,
 - ◆ easier modification,
 - ◆ easier testing and debugging,
 - ◆ better reusability,
 - ◆ easier to set up.
- ❖ A module is any unit of a program that consists of several parts (program, subroutine, package, class, operation, line of code ...)

Modularity principles

- Small modules

Designing with small modules is better.

- Information hiding

Each module should hide its internal structure and processing to the other modules.

- Minimum privileges

Modules should not have access to unnecessary resources.

- Coupling, connascence

Dependence between the modules should be minimized.

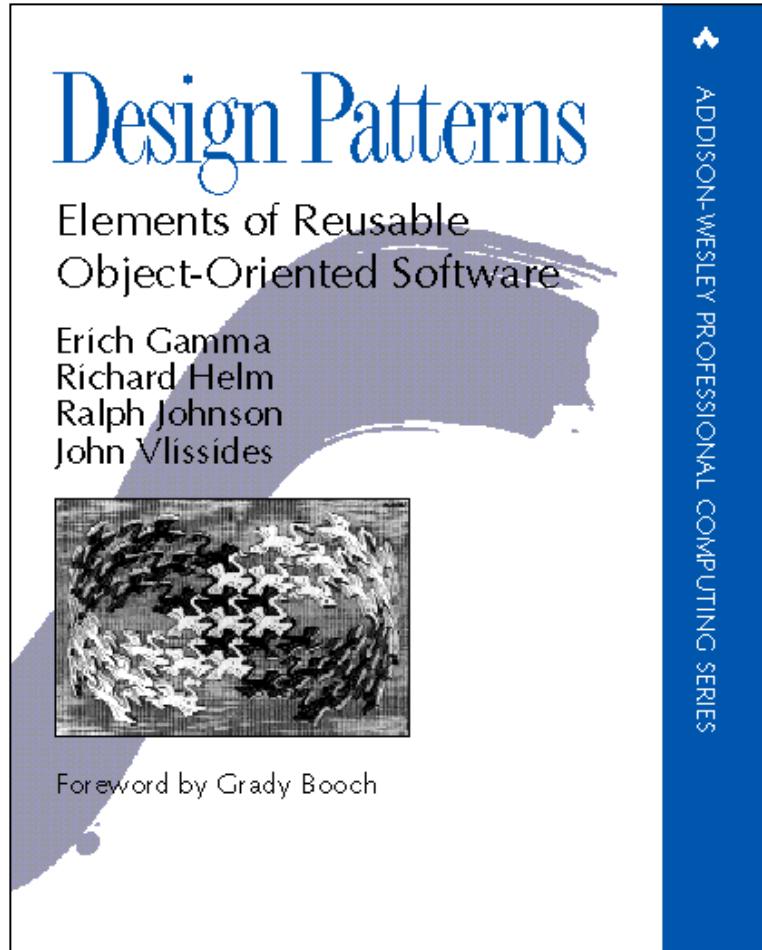
- Cohesion

Cohesion (internal dependency) should be maximized.

Design patterns

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Design patterns



- ❖ Use object modeling techniques to present typical solutions in software design.
- ❖ <http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612>

Design patterns

- ❖ Design patterns are:
 - An elegant solution that is not evident for beginners.
 - Independent of system, programming language or application domain.
 - Extremely successful in real OO systems.
 - Simple, containing only a few classes.
 - Reusable and documented in a general way.
 - Object-oriented, they use OO mechanisms such as classes, objects, generalization and polymorphism.

Design patterns

- ❖ They are defined by:
 - ◆ Pattern name
 - ◆ Problem - Design samples contain a description of the problem for which the sample is appropriate.
 - ◆ Solution - Design patterns give the elements and their relationships that form the solution to the problem.
 - ◆ Implications - Results and consequences of using a design pattern.

Classification of design patterns

- ❖ According to the purpose:
 - ◆ **Creational**
 - They define the process of creating objects
 - ◆ **Structural**
 - They determine the composition of classes and objects
 - ◆ **Behavioural**
 - They determine the way in which classes interact and the distribution of responsibilities.

Creational design patterns

- ❖ **Abstract Factory** groups object factories that have a common theme.
- ❖ **Builder** constructs complex objects by separating construction and representation.
- ❖ **Factory Method** creates objects without specifying the exact class to create.
- ❖ **Prototype** creates objects by cloning an existing object.
- ❖ **Singleton** restricts object creation for a class to only one instance.

Structural design patterns

- ❖ **Adapter** allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
- ❖ **Bridge** decouples an abstraction from its implementation so that the two can vary independently.
- ❖ **Composite** composes zero-or-more similar objects so that they can be manipulated as one object.
- ❖ **Decorator** dynamically adds/overrides behaviour in an existing method of an object.
- ❖ **Facade** provides a simplified interface to a large body of code.
- ❖ **Flyweight** reduces the cost of creating and manipulating a large number of similar objects.
- ❖ **Proxy** provides a placeholder for another object to control access, reduce cost, and reduce complexity.

Behavioral design patterns

- ❖ **Chain of responsibility** delegates commands to a chain of processing objects.
- ❖ **Command** creates objects which encapsulate actions and parameters.
- ❖ **Interpreter** implements a specialized language.
- ❖ **Iterator** accesses the elements of an object sequentially without exposing its underlying representation.
- ❖ **Mediator** allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
- ❖ **Memento** provides the ability to restore an object to its previous state (undo).
- ❖ **Observer** is a publish/subscribe pattern which allows a number of observer objects to see an event.
- ❖ **State** allows an object to alter its behavior when its internal state changes.
- ❖ **Strategy** allows one of a family of algorithms to be selected on-the-fly at runtime.
- ❖ **Template method** defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
- ❖ **Visitor** separates an algorithm from an object structure by moving the hierarchy of methods into one object.

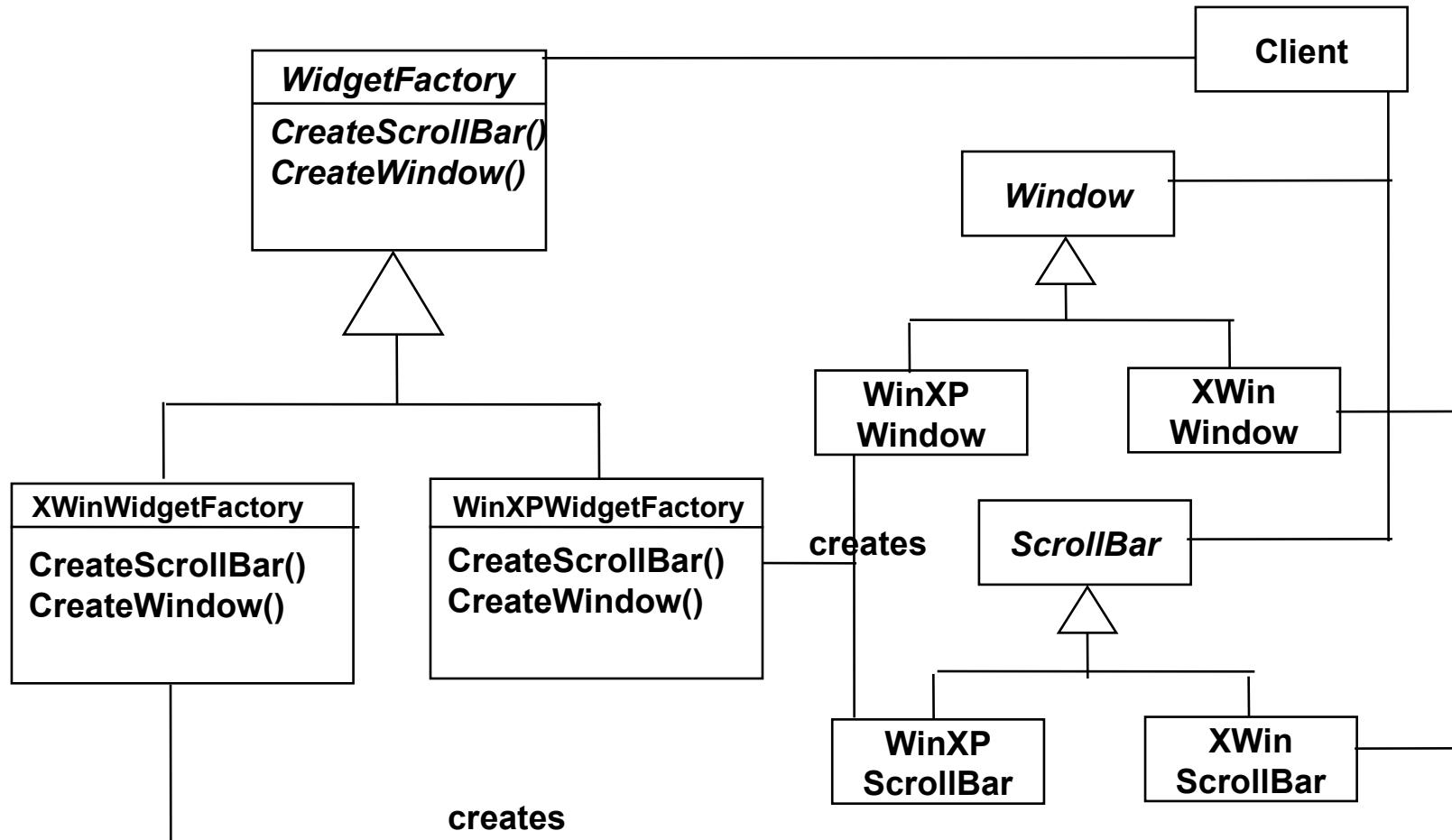
Classification of design patterns

- ❖ Depending on the scope:
 - ◆ Class
 - The pattern deals with classes and their interrelationships, which are determined by inheritance and are static.
 - ◆ Object
 - The pattern deals with relationships between objects that are dynamic and can change at runtime.

Abstract factory

- ❖ The essence of the Abstract Factory Pattern is to "Provide an interface for creating families of related or dependent objects without specifying their concrete classes.".
- ❖ Categories:
 - ❖ Purpose: Creational
 - ❖ Scope: Objects
- ❖ Example:
 - ❖ We want more than one look and feel for the same product. E.g. Windows XP and X-Windows.
- ❖ We want to avoid hard-coded widgets of the interface.

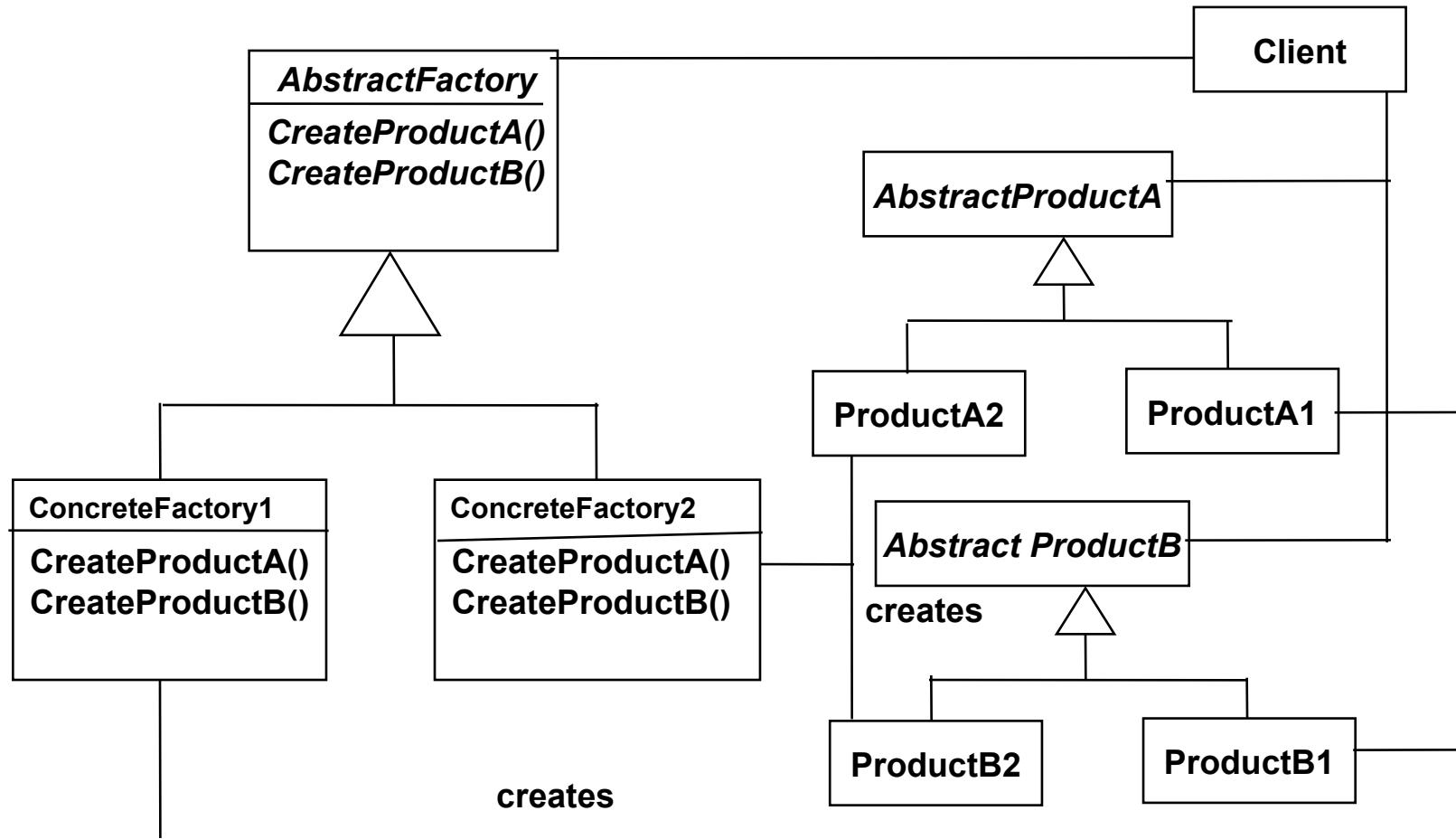
Abstract factory - example



Abstract factory - usage

- ❖ The pattern is used in the following situations:
 - ◆ The system must be configurable with different product families.
 - ◆ The family of dependent products is designed to be used together, and this restriction must be ensured.
 - ◆ We want to provide a product library, but we only want to disclose the interface and not the implementations.

Abstract factory - struktura



Abstract factory - participants

- ❖ **AbstractFactory**
 - ◆ Declares an interface for operations that create abstract product objects.
- ❖ **ConcreteFactory**
 - ◆ Implementing operations to create abstract product objects.
- ❖ **AbstractProduct**
 - ◆ Declares an interface for the product object type.
- ❖ **ConcreteProduct**
 - ◆ Specifies the product object to be created with the associated ConcreteFactory and implements the AbstractProduct interface.
- ❖ **Client**
 - ◆ Uses only interfaces declared with AbstractFactory and AbstractProduct classes.

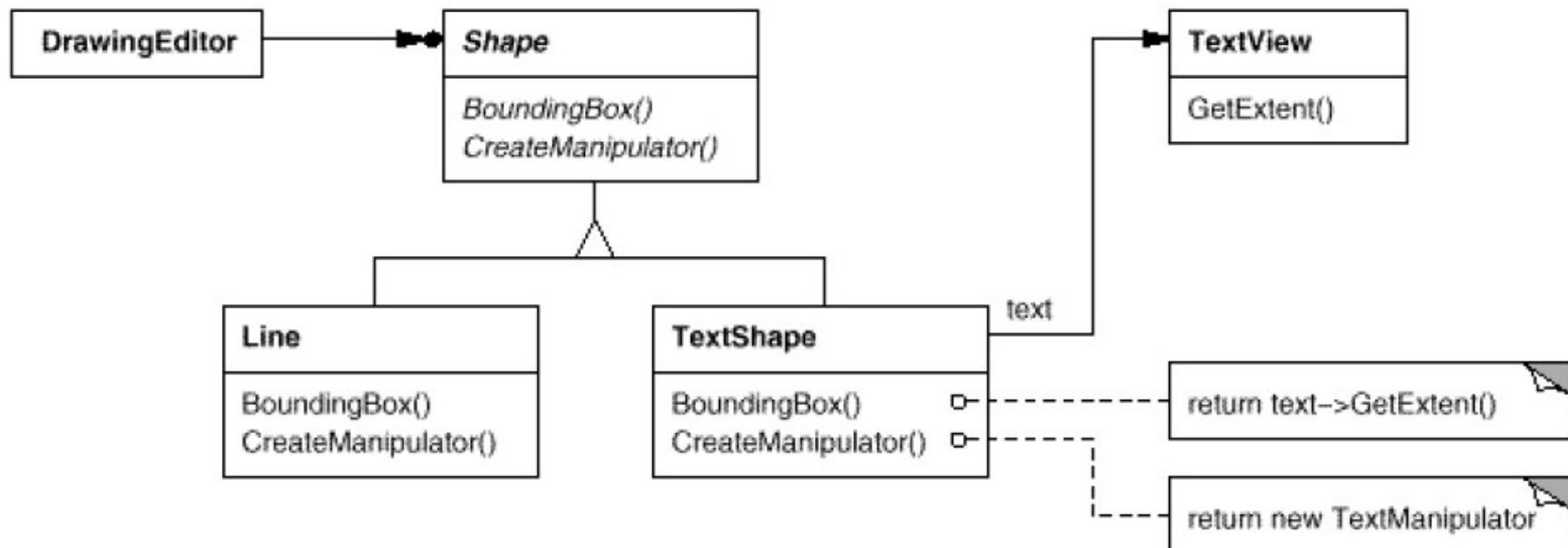
Abstract factory - implications

- ❖ Client isolation from actual class implementations.
- ❖ Clients manipulate instances through an abstract interface.
- ❖ Product class names do not appear in the client code.
- ❖ It makes it easy to swap product families - only by using the second ConcreteFactory class, which only appears once in the client code.
- ❖ Promotes product consistency across families.
- ❖ Extending with new products is difficult, requiring the extension of the AbstractFactory class and all ConcreteFactory classes.

Adapter

- ❖ The adapter design pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients.
- ❖ Categories:
 - ◆ Purpose: Structural
 - ◆ Scope: Class
- ❖ Example:
 - ◆ Using an existing class to print text in an image editor with incompatible class interface.
- ❖ We want to make it possible to use an existing class, even if its interface is not appropriate.

Adapter - example

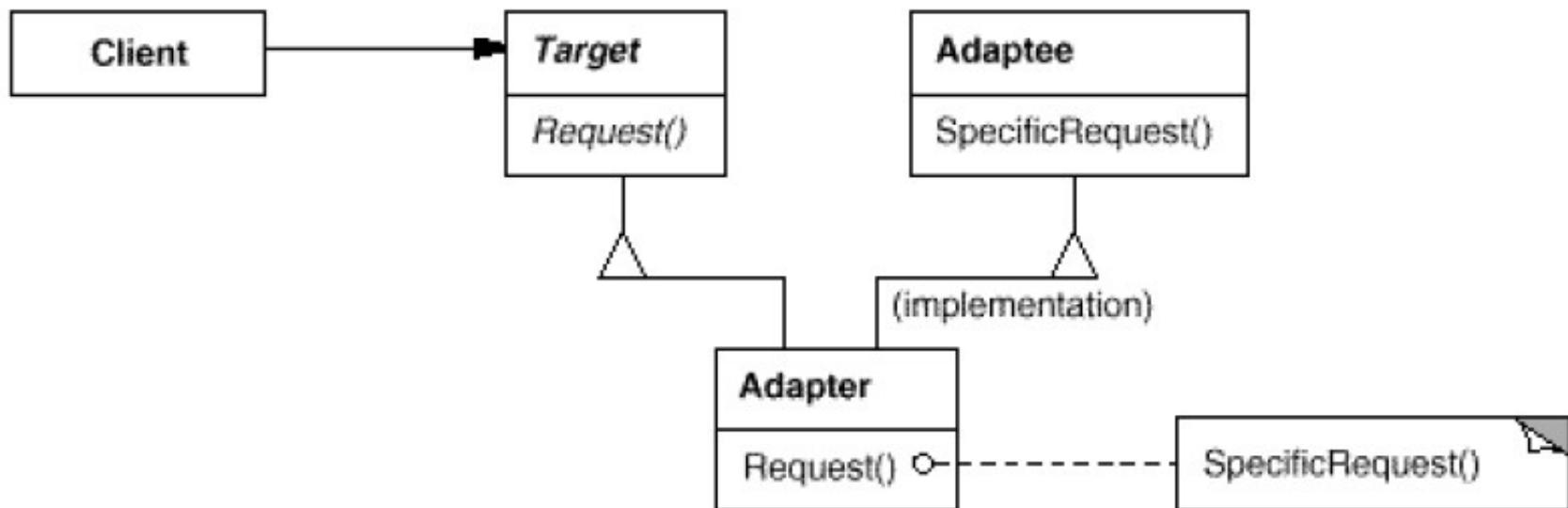


Adapter - usage

- ❖ The pattern is used in the following situations:
 - ◆ We want to use an existing class, but its interface does not match the one we need.
 - ◆ We want to build a reusable class that can be used by a wide variety of other classes (including unknown ones).

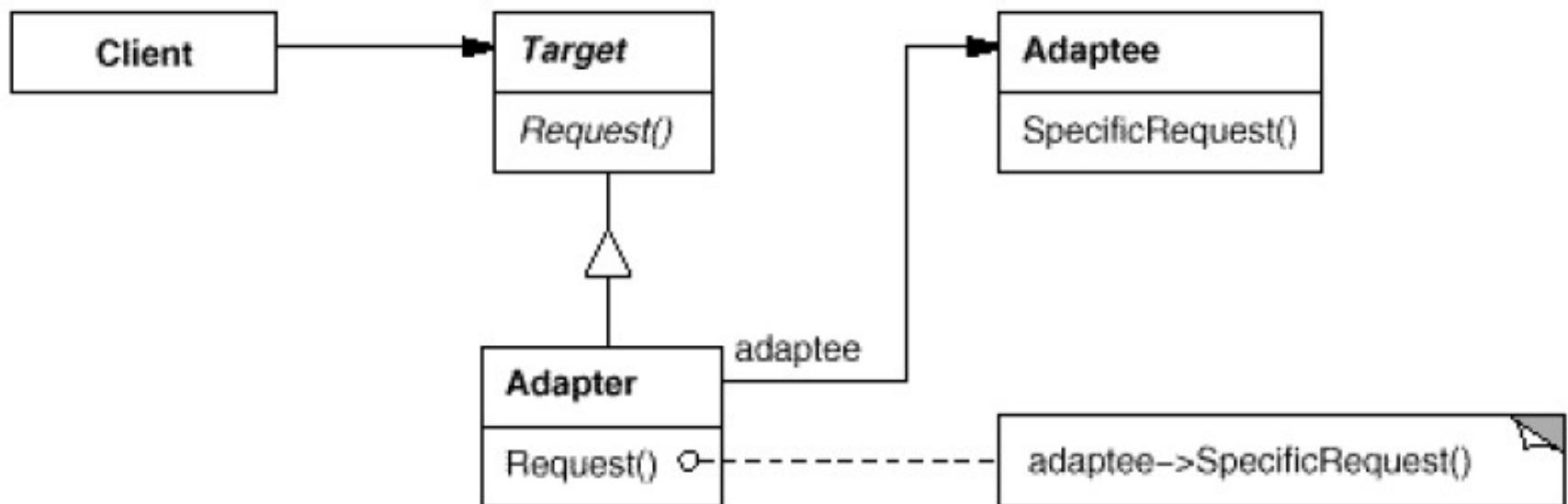
Adapter - structure

❖ A: using multiple inheritance



Adapter - structure

❖ B: with object composition



Adapter - participants

- ❖ **Target**
 - ◆ Defines the domain-specific interface used by the Client object.
- ❖ **Client**
 - ◆ Collaborates with objects using the Target interface principle
- ❖ **Adaptee**
 - ◆ Specifies the existing interface that needs to be customized.
- ❖ **Adapter**
 - ◆ Translates the Adaptee object interface to the Target interface

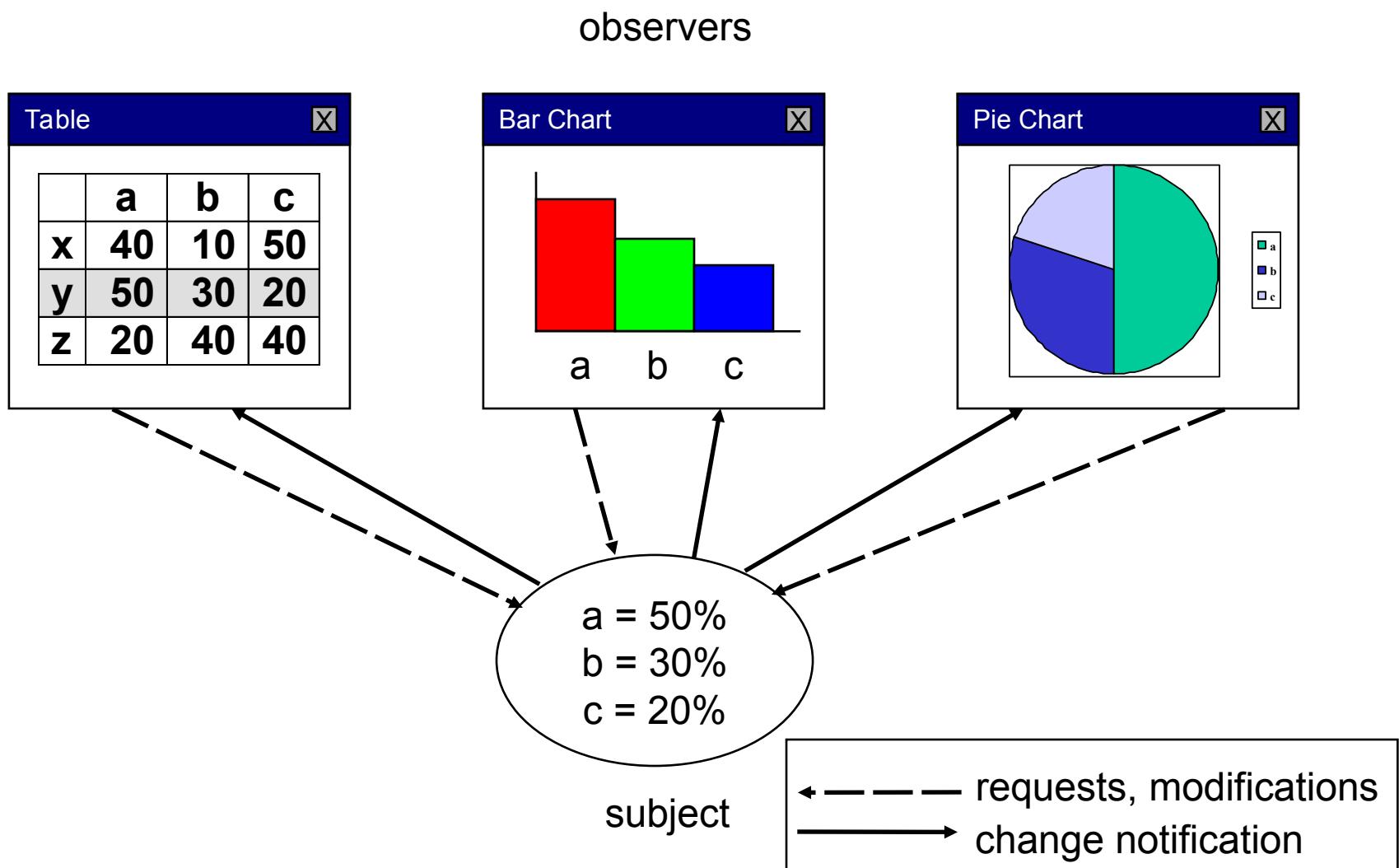
Adapter - implications

- ❖ Adaptee interface is adapted to suit Target.
- ❖ An adapter can override the properties of an Adaptee class if it inherits it.
- ❖ In the case of object composition, the adapter can adapt several Adaptee classes at a time.
- ❖ A two-way adapter is possible.

Observer

- ❖ The Observer design pattern specifies a mechanism for linking multiple objects so that all objects are notified of a change of one of the objects.
- ❖ Categories:
 - ❖ Purpose: behavioral
 - ❖ Scope: objects
- ❖ Example:
 - ❖ View the same data in spreadsheets and graphs.
- ❖ We want the number of dependent objects to be unlimited and can change dynamically.

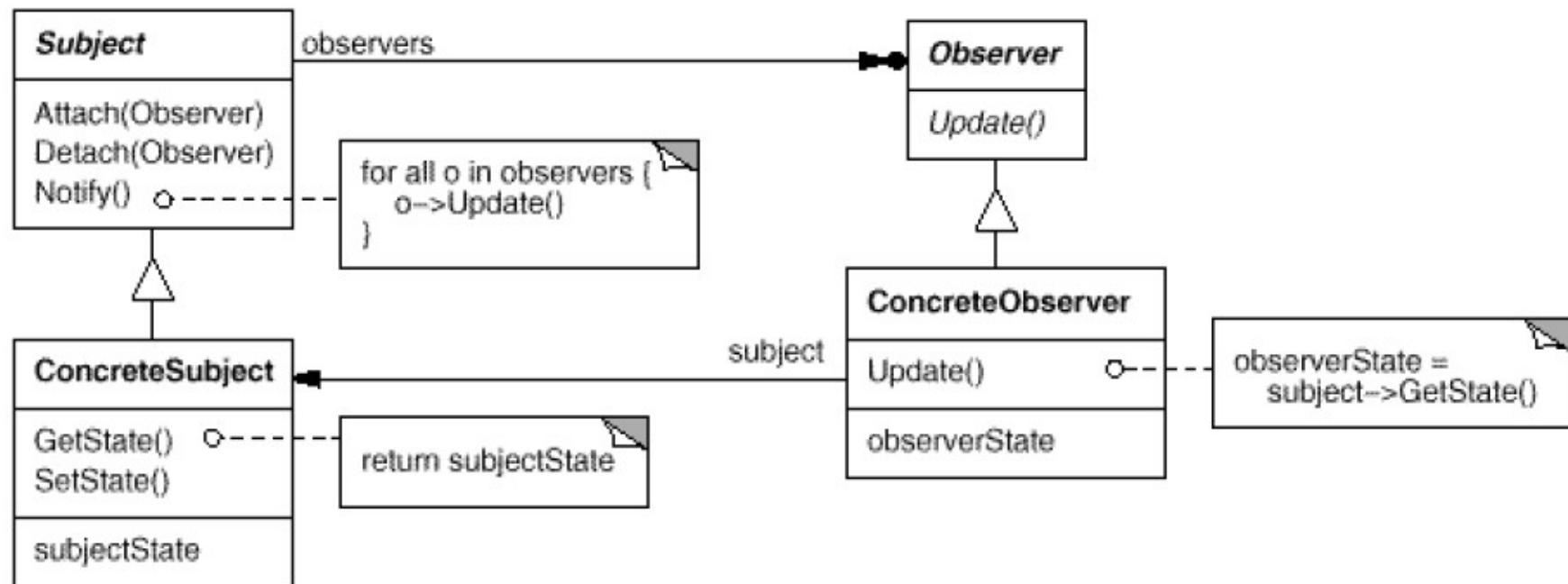
Observer - example



Observer - usage

- ❖ The pattern is used in following situations:
 - ◆ A one-to-many dependency between objects should be defined without making the objects tightly coupled.
 - ◆ It should be ensured that when one object changes state an open-ended number of dependent objects are updated automatically.
 - ◆ It should be possible that one object can notify an open-ended number of other objects.

Observer - structure



Observer - participants

- ❖ **Subject**
 - ◆ Knows the open ended Observer objects, ki jih je lahko poljubno mnogo.
 - ◆ Nudi vmesnik za priklapljanje in odklapljanje objektov Observer.
- ❖ **Observer**
 - ◆ Specifies an interface for informing objects about changing a Subject object.
- ❖ **ConcreteSubject**
 - ◆ Contains a state that is relevant to ConcreteObserver objects.
 - ◆ Sends information about the change when it occurs.
- ❖ **ConcreteObserver**
 - ◆ Contains a reference to the ConcreteSubject object
 - ◆ Keeps a state that must be consistent with the state of the ConcreteSubject object.
 - ◆ It implements an update interface to maintain a consistent state with the ConcreteSubject object.

Observer - implications

- ❖ Abstract and minimal association between Subject and Observer objects.
 - ◆ Subject is not aware of the actual class of the Observer.
 - ◆ Subject and observer may belong to different layers due to their poor connectivity.
- ❖ Support of dispersed broadcasting.
 - ◆ No need for mutual knowledge.
 - ◆ The message is sent to all interested (subscribed) observers.
 - ◆ Ability to dynamically add or remove observers.
- ❖ A seemingly innocuous subject change can trigger an avalanche of changes to observers and dependent classes!
- ❖ Poorly defined class dependencies can lead to false updates.

Design patterns - general

- ❖ There is a large number of design patterns (23 well known published in the Design Patterns book).
- ❖ Everyone can define/propose his own additional design patterns.
- ❖ The selected patterns shown are presented to give a better idea of the design patterns in general.
- ❖ Design patterns are useful, and knowing them can make programming easier and help improve quality.

Documenting

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Documenting: component design

- ❖ The result of component design activity is a component design or a detailed design.
- ❖ A component design must define
 - ◆ The internal structure of all components
 - ◆ Their dependence (interfaces)
 - ◆ Their structure (operations, attributes, inheritances, compositions)
 - ◆ Processing methods
 - ◆ Algorithms,
 - ◆ Data structures,
 - ◆ Properties (functional and non-functional - throughput, reliability ...)

Interface specification

❖ *Syntax*

- ◆ It defines the elements of the communication medium and how they are grouped into messages.

❖ *Semantics*

- ◆ Specifies the meaning of messages

❖ *Pragmatics*

- ◆ Specifies how messages are used to perform tasks.

Review!

- ❖ Similar to system design:
 - ◆ emphasis on the internal structure of components (modules) and interfaces.

Software engineering

Testing

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

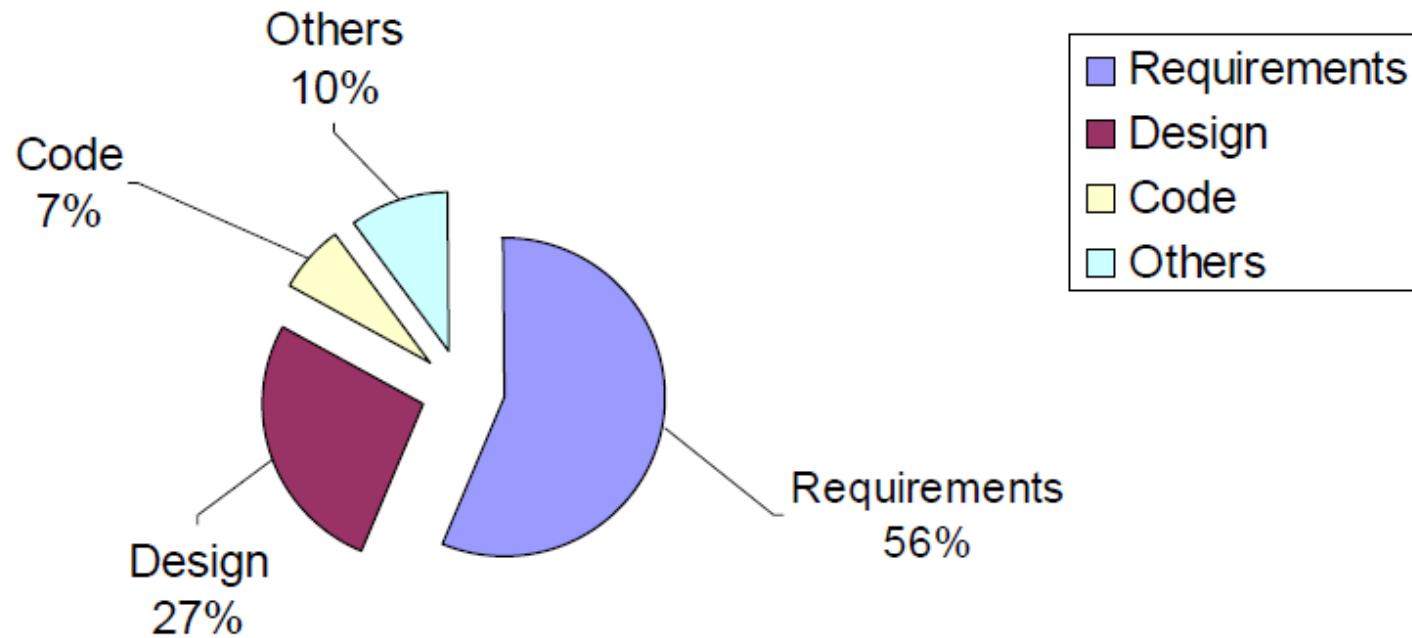
When it does not work...

Top 24 replies by programmers when their programs don't work:

- 24. "It works fine on MY computer"
- 23. "Who did you login as ?"
- 22. "It's a feature"
- 21. "It's WAD (Working As Designed)"
- 20. "That's weird..."
- 19. "It's never done that before."
- 18. "It worked yesterday."
- 17. "How is that possible?"
- 16. "It must be a hardware problem."
- 15. "What did you type in wrong to get it to crash?"
- 14. "There is something funky in your data."
- 13. "I haven't touched that module in weeks!"
- 12. "You must have the wrong version."
- 11. "It's just some unlucky coincidence."
- 10. "I can't test everything!"
- 9. "THIS can't be the source of THAT."
- 8. "It works, but it's not been tested."
- 7. "Somebody must have changed my code."
- 6. "Did you check for a virus on your system?"
- 5. "Even though it doesn't work, how does it feel?"
- 4. "You can't use that version on your system."
- 3. "Why do you want to do it that way?"
- 2. "Where were you when the program blew up?"
- 1. "I thought I fixed that."

Defect distribution statistics

Defect Distribution in Software Processes "SW Life Cycle"



Quality assurance

- ❖ Each step of the software process must be completed by review or testing.
 - ◆ In this phase, we determine if the step results meet the step's input requirements (verification) and that they comply with the specification requirements (validation) as a guide to assessing user compliance.
 - ◆ Quality assurance begins early in the software process - from the very beginning stages.
 - ◆ The intensity of quality assurance increases in the implementation phase (coding), with source code review and verification.
- ❖ The peak of testing activities is in the testing phase with validation and system testing.

Verification and validation

❖ Verification

- ◆ The individual components are tested for compliance with the component specifications – the component design.
- ◆ The individual components are integrated into the software system and tested according to the system specifications - the system design.

❖ Validation

- ◆ Executed by following a pre-prepared plan that follows software requirements specifications (SRS).
- ◆ Tests the compliance of the system with the requirements (functional and non-functional, also taking into account implicit or "self-evident" requirements).

Recensions / Reviews

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Recensions / Reviews

- ❖ Presentation of expert opinion (assessment), judgment on some (new) work, especially in terms of quality.
- ❖ In the software process, as the final part of each task, intended for quality assurance or verification and validation.
- ❖ In the implementation step we are talking about code review (code inspection).

Recension / Review

- ❖ It is an activity in which one or more people systematically scan the documentation for the purpose of finding possible errors.
- ❖ Typically, a review involves a meeting with the authors, although reviewers can also do the review individually or separately.

Constructiveness and positivity

In many organizations, reviews are unpopular events:

- ◆ The author often feels that the purpose of the review is to challenge his opinion.
- ◆ Often there is no distinction between important and irrelevant issues.
- ◆ Other participants may use the review to demonstrate their "better" abilities.
- ◆ ...
- ❖ Reviews should not be directed towards the author, they must be constructive and positive.

The purpose of the reviews

- ❖ Ensuring the quality of a peer-reviewed product (eg source code). Viewers can effectively find various discrepancies, ranging from deviation of business processes to simple errors in source code.
- ❖ A teaching tool for developers to share knowledge of quality improvement techniques (eg source code), consistency, ease of product maintenance, and use of different approaches.

Positive reviewers approach

Ask instead of gravel.

- ◆ E.g. instead of "You did not follow the standards here!" use "What is the reason for using the approach used?"

Avoid "Why" questions.

- ◆ E.g. instead of "Why didn't you follow the standards here?" use "What is the reason for deviating from the standards ..."

Don't forget the praise.

- ◆ It is also important to notice the positive, e.g. creative solutions. This lets the author know that we see his work more broadly than just a series of mistakes.

Don't forget the praise.

- ◆ The coding standards must be accepted as an agreement between the developers. If it is an issue that has not yet been agreed, it is appropriate to include it further in the standard.

The topic of the review should be the work and not the author.

- ◆ The purpose of the review is to contribute to the quality and not to determine the attributes and abilities of the author.

Be aware that there is more than one possible (good) solution.

- ◆ Although the author did something differently than you would, this is not necessarily wrong.

A positive approach from the authors

Be aware that the peer-reviewed work is not you.

- ◆ Development is a creative process and it is normal for you to be attached to your work. However, reviewers do not want to tell you that you are not a good developer (or person). Their mission is to point out the shortcomings and possibilities of better solutions. Even if reviewers do not do their job well, do not take a defensive position and listen to constructive comments (and overhear any attacks).

Make a list of things that reviewers will focus on.

- ◆ Use the list to reconsider your work and correct any errors found. This will not only reduce the number of errors found, but also shorten the audit time (to the delight of all involved).

Help maintain standards.

- ◆ Offer to supplement standards (eg coding) for what has been agreed and not included in existing standards. In this way, the documented solution will help with the works and reviews that will follow.

Documenting reviews

For reviews that are a list of comments, follow these guidelines:

❖ **Start with a resume and be positive..**

- ◆ This shows the author your disposition and softens the comments that follow. The comment should contain positive messages to the author, as every work contains something good, which is worth mentioning (even if it is only syntactic correctness). A positive attitude is essential.

❖ **Use electronic mechanisms to record comments.**

- ◆ E.g. copy to word and use word functions for commenting. This way you are not limited by the length of the comments and you have the opportunity to clarify your position and make the comments easier to formulate in a question form.

❖ **Make an agreement that the author does not need to answer all the questions.**

- ◆ Good reviews also contain thought-provoking questions such as: "Wouldn't it be better to use the X design pattern?". Such questions do not need to be answered as they are intended solely to stimulate thinking. Such issues do not have a significant impact on the existing solution but have a positive long-term impact on product quality.

Principles

- ❖ Reviews should be made for the most important documents (requirements specification, system and component design plans, test plans) and source code.
- ❖ An effective review team should consist of 2 to 5 members and include experienced developers.
- ❖ Participants should prepare for the review meeting.
- ❖ Do not review in-completed documents.
- ❖ Avoid discussing how to make corrections.
- ❖ Don't rush - when revising the source code, it's a good speed of 200 lines per hour.
- ❖ The review should not exceed two hours in total and four hours per day.
- ❖ Revisions should be made if more than 20% of the work is changed.
- ❖ The management should not be included in the reviews as this has a negative impact on the openness of other reviewers.
- ❖ No one should be offended.

Reviews vs testing

- ❖ Reviews and testing are based on different ways of verifying a product.
- ❖ Reviews allow you to find bugs that affect maintenance possibilities and performance.
- ❖ Testing makes it possible to look for errors in complex solutions (deep code error) and have clear consequences.
- ❖ The chances of error are smaller when both review and testing are used.
- ❖ Reviews should take place before any testing.

Source code reviews

❖ Guidelines:

- ◆ http://www.oualline.com/talks/ins/inspection/c_check.html
- ◆ <http://www.literateprogramming.com/Baldwin-inspect.pdf>
- ◆ <http://www.scribd.com/doc/4957605/Code-Review-Rules>
- ◆ ...

Use tools for (static) source code analysis.

Testing

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Additinal literature: Roger Pressman, Software Engineering. A Practitioner's Approach, McGraw Hill, 2000, poglavji 17 in 18.

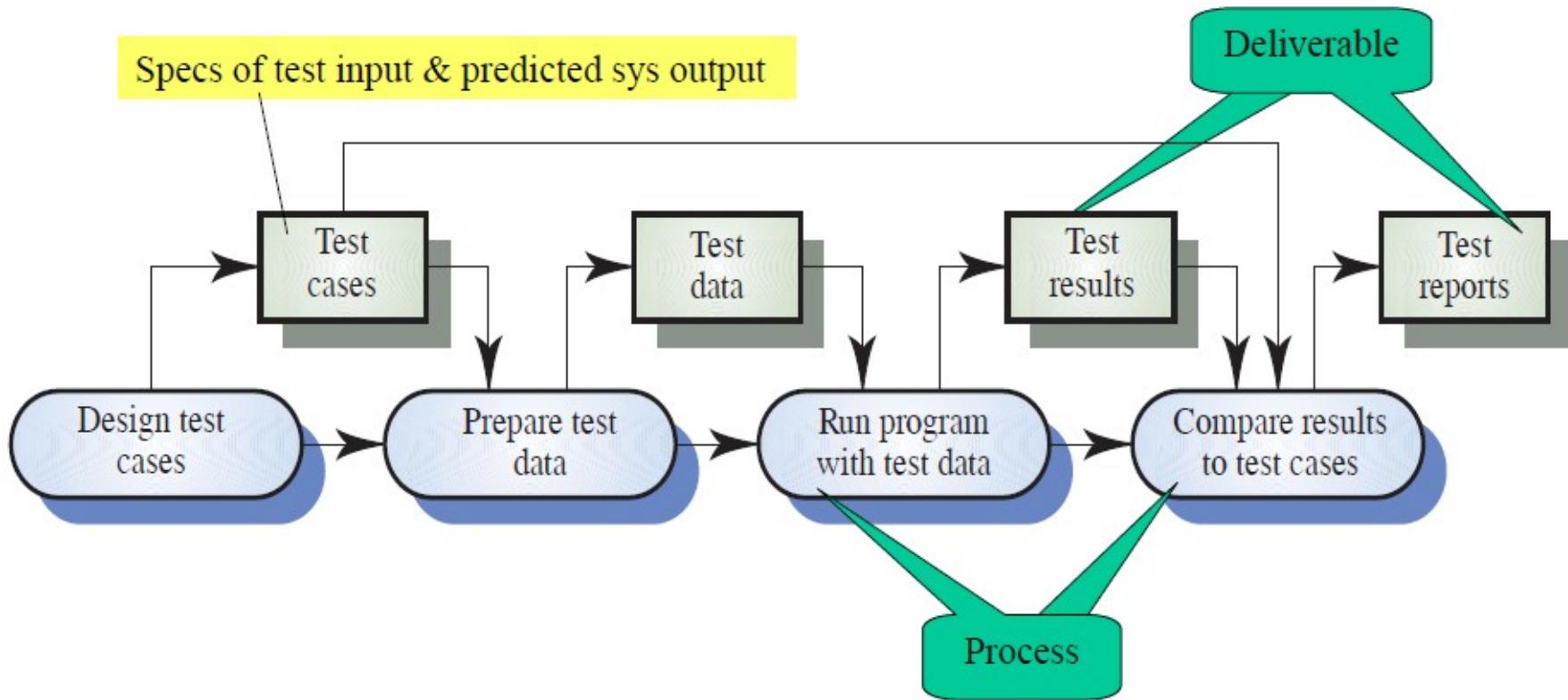
Testing ☺

- ❖ *To the optimist, the glass is half full.*
- ❖ *To the pessimist, the glass is half empty.*
- ❖ *To the good tester, the glass is twice as big as it needs to be.*

Testirng

- ❖ Testing is the process of executing a software with the aim of finding an error before passing it on to end users.
- ❖ The tester performs a series of test cases in order to destroy the proper functioning of the program.
- ❖ A good test case is one that is likely to reveal an error not yet discovered.
- ❖ A test is successful if a previously unknown error is found.

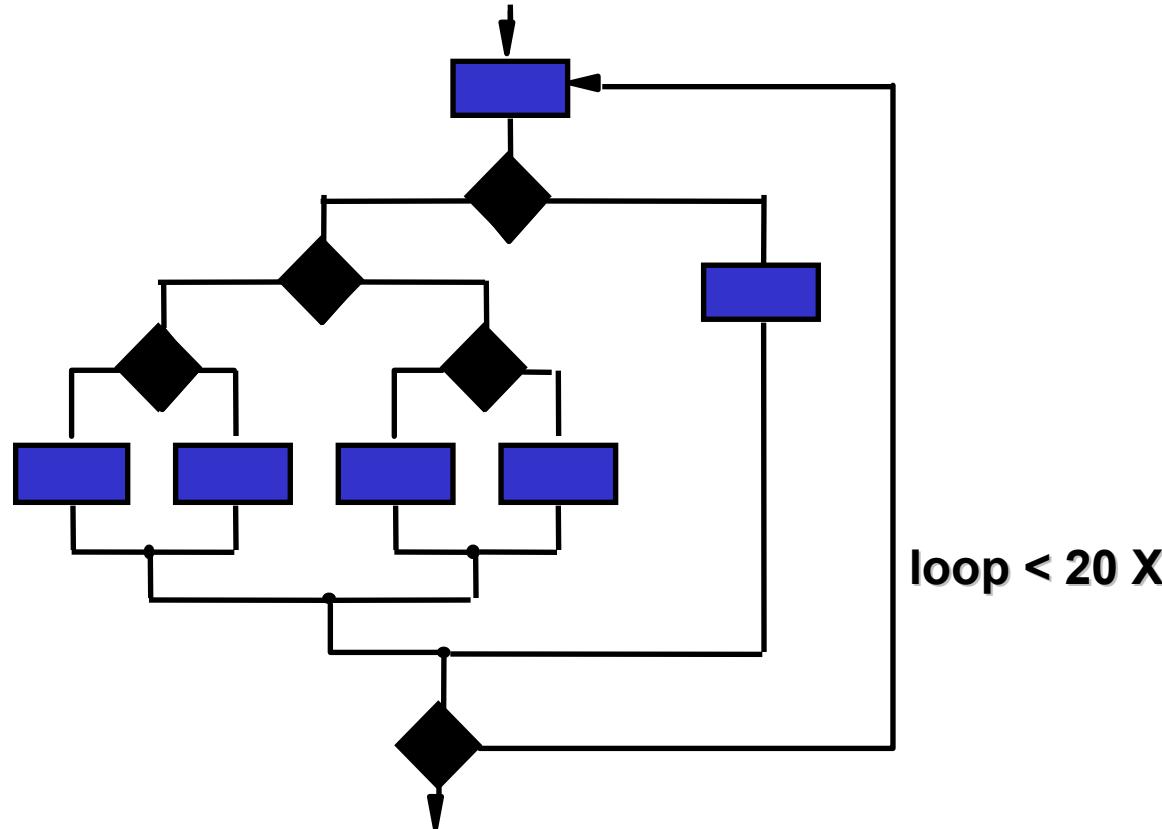
Testing



Testing principles

- ❖ All test cases should be related to the requirements.
 - ◆ The worst mistakes are those that cause make program not to fulfill the requirements.
- ❖ Testing should be scheduled before it begins.
 - ◆ Test planning can begin as soon as the requirements specifications are finalized.
- ❖ When testing, Law 80-20 (Pareto principle) applies.
 - ◆ 80% of all errors detected during testing originate from 20% of software components. These components need to be tested so much more thoroughly.
- ❖ Comprehensive testing is not possible.
 - ◆ Even for smaller software components, the number of possible paths for running it may be too large to test all the options.
- ❖ Independent testers perform the most effective testing
 - ◆ The developer who made the software is not the preferred tester because of a conflict of interest.

Exhaustive testing



- ❖ There are 10^{14} possible routes.
- ❖ If it took 1ms to run each one, the program would run 3170 years!
- ❖ Selective testing is essential!

Who performs the testing

❖ SW developer:

- ◆ He understands the system well.
- ◆ The main interest is the timely delivery of the product.
- ◆ It is constructive and gentle when tested against the system.

❖ Independent tester:

- ◆ He does not know the system and has yet to know it.
- ◆ The main interest is quality assurance.
- ◆ It is destructive and will look for ways to break the system.

Testability

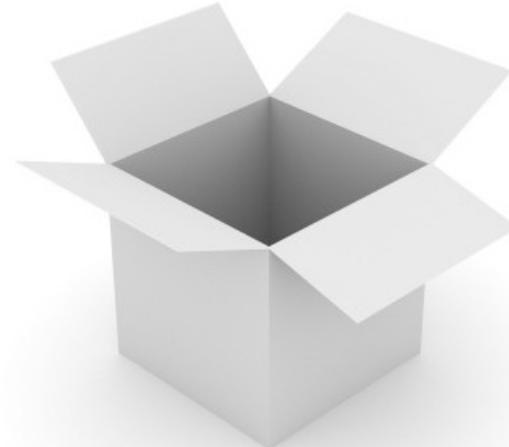
- ❖ Testability is a property that tells how easy it is to test a program.
- ❖ Good SW testability is one of the design goals.
- ❖ Testability depends on several other features of the SW:
 - ◆ Operability - The SW can be run smoothly.
 - ◆ Observability - the SW provides clear insight into the details of SW execution.
 - ◆ Controlability - the degree to which testing can be automated and optimized.
 - ◆ Decomposability - testing can be focused on a single module.
 - ◆ Simplicity - the simplicity of the SW allows for more thorough testing.
 - ◆ Stability - the rate of change during testing.
 - ◆ Understandability - understanding the operation of the SW, including documentation.

Testing methods

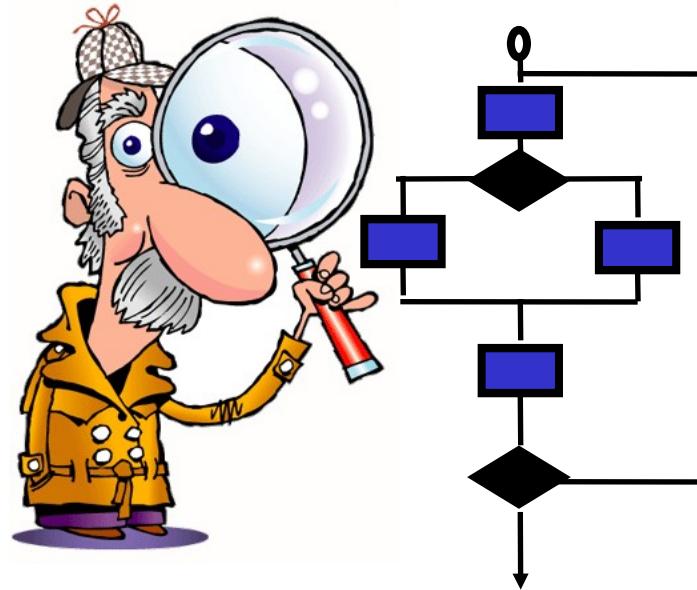
doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Testing methods

- ❖ Structural testing
(white-box testing)
 - ◆ Testing is based on knowledge of the internal structure of the program.
- ❖ Behavioral testing
(black-box testing)
 - ◆ Testing is based on the (declared) external properties of the system.



Structural testing (white-box)



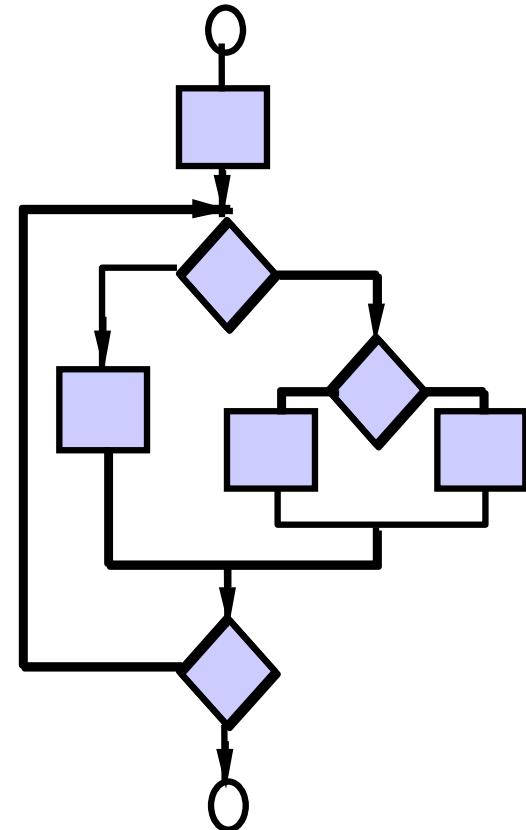
- ❖ The goal is to ensure that each condition has been satisfied at least once and every part of the source code has been executed at least once.

The overall SW structure

- ❖ When testing, check the overall structure of the SW - all paths of the execution.
 - ◆ Logical errors and false assumptions are inversely proportional to the probability of path execution.
 - ◆ We often believe that a certain path is unlikely to be taken during execution. Reality is often contrary to intuition.
 - ◆ Typing errors occur at random. They are also likely to be present on untested routes.

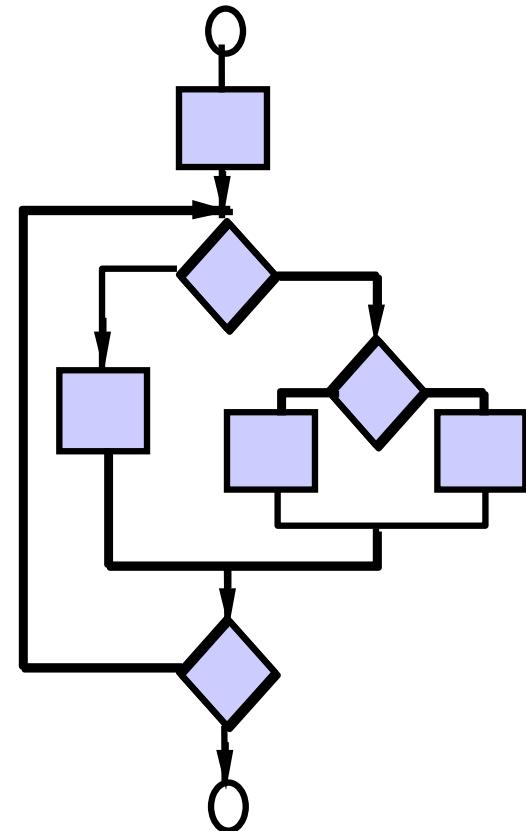
Basis Path Testing

- ❖ A **basis path** is a path through a SW that contains at least one new piece of program code or a new condition.
 - ◆ At least one part of the path is not contained in other independent routes.
- ❖ The number of independent paths depends on the **cyclomatic complexity** - a measure of the logical complexity of the program.



Basis Path Testing

- ❖ Determining cyclomatic complexity $V(G)$ based on execution graph (activity diagram) G :
 - ◆ $V(G) = \text{number of simple decisions} + 1$
 - ◆ $V(G) = \text{number of contained areas of graph} + 1$
 - ◆ For the given example:
 $V(G) = 3 + 1 = 4$



Basis Path Testing

- ❖ Determination of independent paths.

$$V(G) = 4 ;$$

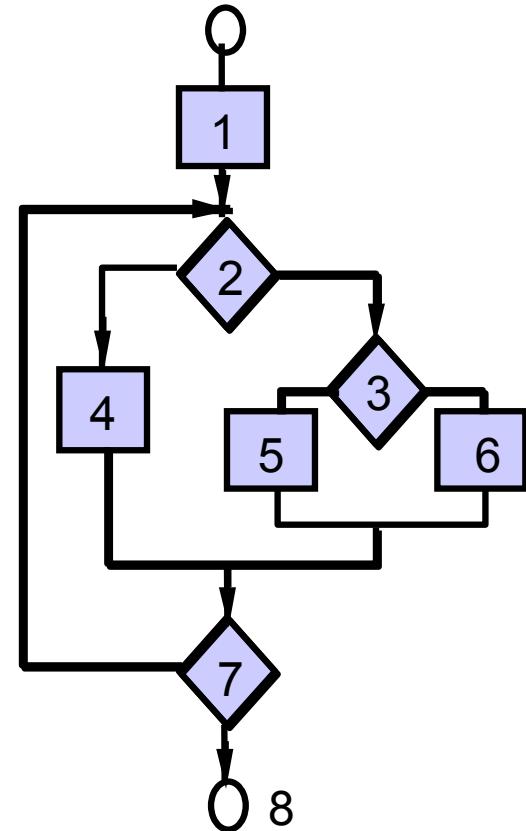
we can define 4 basic paths::

- ◆ Path1: 1,2,4,7,8
- ◆ Path2: 1,2,3,5,7,8
- ◆ Path3: 1,2,3,6,7,8
- ◆ Path4: 1,2,4,7,2,4,7,8

- ❖ Each additional path is a combination of existing routes and is therefore not independent.

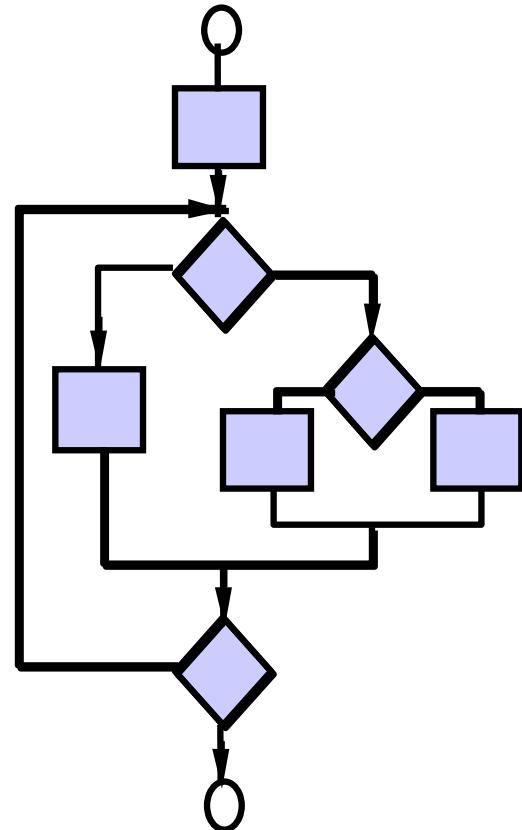
- ◆ 1,2,3,6,7,2,4,7,8 -combination of paths 3&4

- ❖ Basic paths define test cases for structural testing.



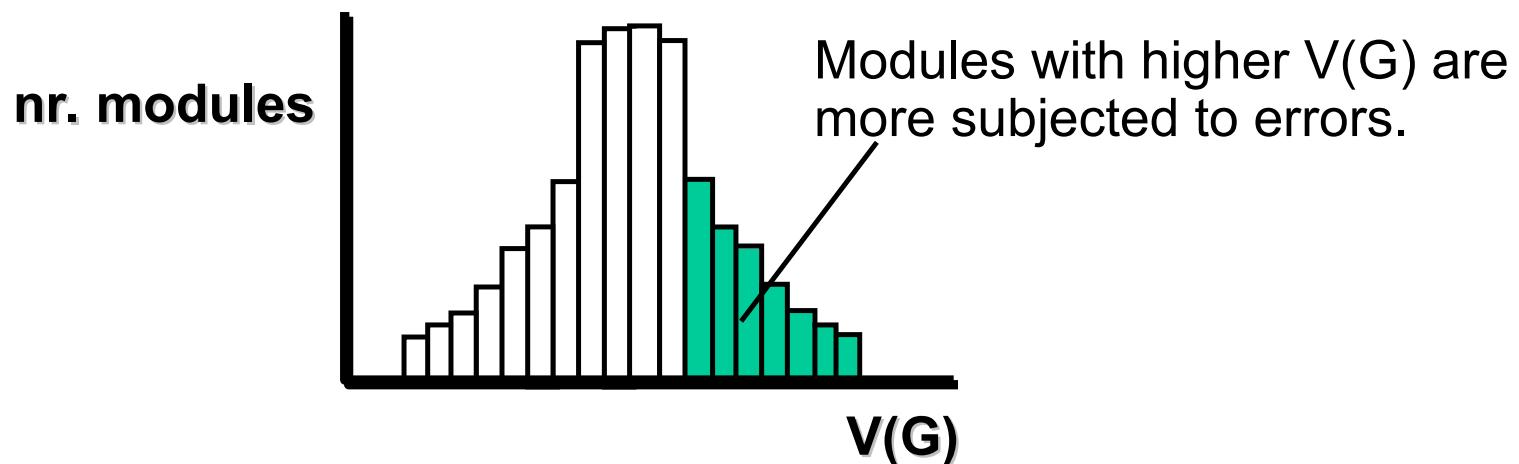
Basis Path Testing

- ❖ Basic paths can be defined without using a graph, but graph makes it easier to follow the paths.
 - ❖ Cyclomatic complexity can be determined by counting simple logical decisions. Complex decisions are considered two or more simple ones.
 - ❖ Basic path testing is especially important for critical modules.

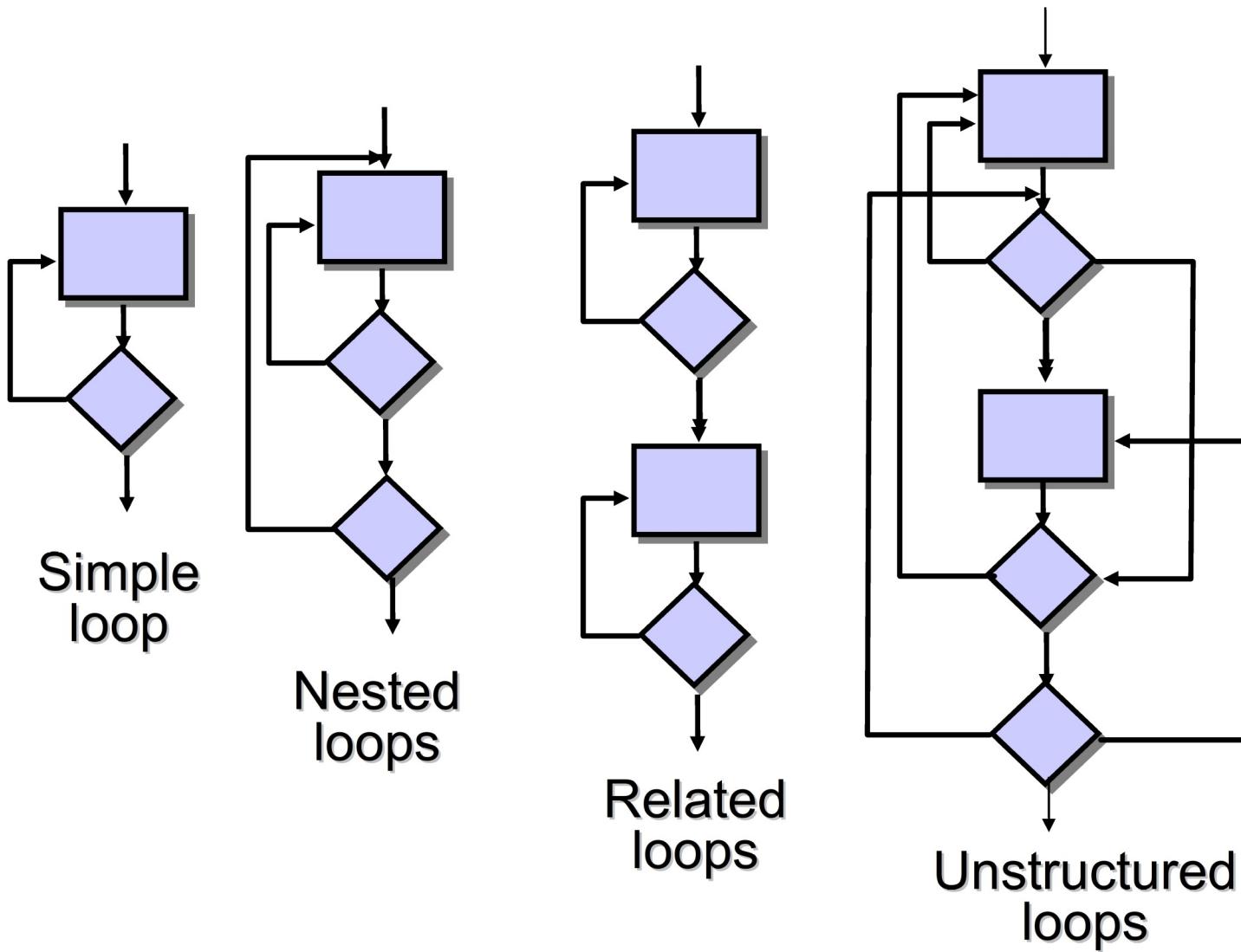


Cyclomatic complexity

- ❖ Industry research shows that the likelihood of failure is positively related to the cyclomatic complexity of $V(G)$.



Testing of loops



Testing of simple loops

- ❖ A set of test cases should include:
 - ◆ Leaving out the loop
 - ◆ A single pass through the loop
 - ◆ Two passes through the loop
 - ◆ m passes through a loop where $m < n$
 - ◆ $n-1, n, n + 1$ passes through the loop

n - the maximum number of loop passes allowed

Testing of nested loops

- ❖ Start with the innermost loop. All other loops should be executed with the least number of iterations.
- ❖ Test the loop for $\min + 1$, $\max - 1$ and the typical number of iterations.
- ❖ Move to one level a more external loop and test it the same way. Continue this way until you have completed the outermost loop.

Testing of related loops

- ❖ If the loops are independent, then treat each one as a simple loop
- ❖ If the loops are not independent, treat them as nested loops.
 - ◆ Dependency example: The final counter of one loop affects the initialization of the next loop.

Testing of conditions

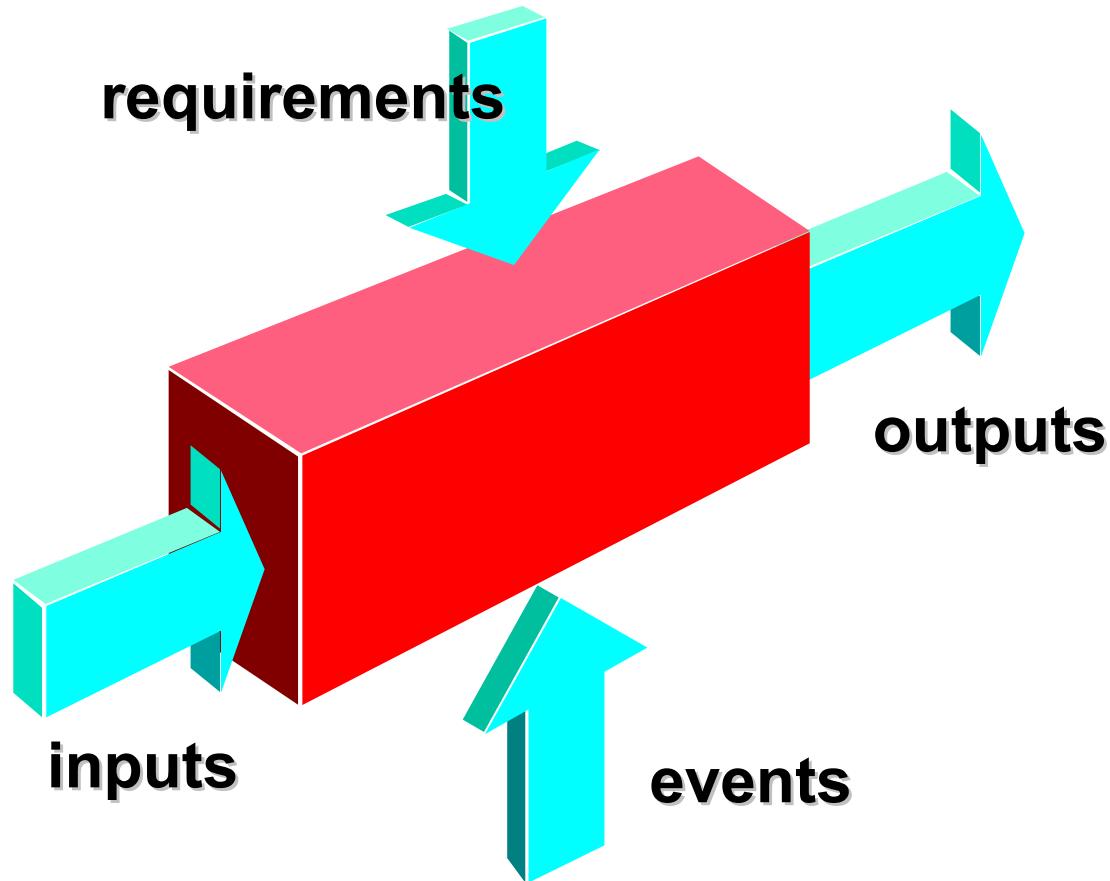
- ❖ Each decision is made based on some condition.
- ❖ Condition testing is based on the selection of such test cases to verify the correctness of the decisions against the decision parameters.
- ❖ The following errors are possible:
 - ◆ Logical operator error (inappropriate, missing, redundant)
 - ◆ Logical variable error
 - ◆ Error in brackets of logical expression
 - ◆ Error in relational operator ($>$, $<$, \leq , \geq , $=$, \neq)
 - ◆ Error in arithmetic operator
- ❖ The test cases for testing the conditions partially overlap with the test cases for testing the basic paths.

Behavioral testing (black-box)



- ❖ Unlike white-box, black-box testing is used in the later stages of the testing process.
- ❖ It does not focus on the control flow but the processed information.
- ❖ Test case planning is based on the following questions:
 - ◆ How to verify functional correctness?
 - ◆ How to check system behavior and throughput?
 - ◆ What input form good test cases?
 - ◆ Is the system sensitive to certain input values?
 - ◆ What are the boundaries of data classes?
 - ◆ What range of data can the system tolerate?
 - ◆ What impact do specific combinations of data have on the performance of the system?

Behavioral testing (black-box)



Equivalence partitioning

- ❖ Equivalence partitioning is a procedure for determining the classes of input parameters from which to select test data.
- ❖ An equivalence class is a group of valid or invalid input conditions.
- ❖ Depending on the input conditions, we define the equivalent classes as follows:
 - ◆ The input condition is field-specified - specifies one valid and two invalid equivalent classes.
 - ◆ The input condition is a specific value - it specifies one valid and two invalid equivalent classes.
 - ◆ The input condition is determined by the membership of the set - it specifies one valid and one invalid equivalent class.
 - ◆ Boolean - Specifies one valid and one invalid equivalent class.

Boundary value analysis

- ❖ Boundary value analysis - BVA) leads to the selection of test cases based on boundaries of input data.
 - ◆ Eg: the input condition is determined by the region bounded by a and b, - test cases should use the values a and b and the lower and higher values.
 - ◆ ...
- ❖ Testing with a BVA is not only about the input but also the output!

Other behavior testing techniques

- ❖ Error guessing methods
- ❖ Decision table techniques
- ❖ Cause-effect graphing

Testing strategies

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

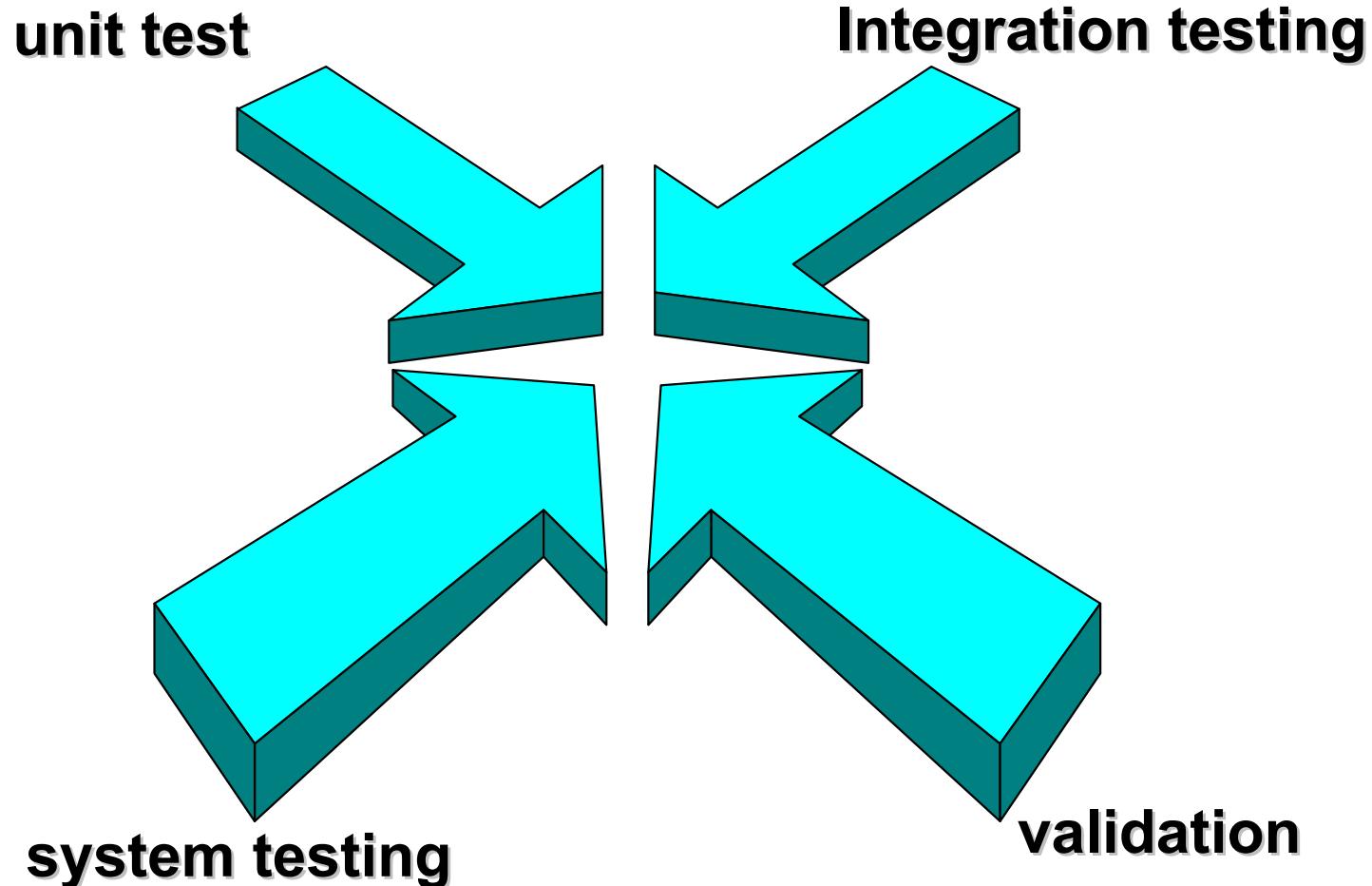
Testing strategies

- ❖ Several testing strategies have been proposed in the literature. They have the following characteristics in common:
 - ◆ Testing is from the bottom up. It starts at the component level (eg classes, objects) and continues with the consideration of ever larger modules and ends with testing of the entire system.
 - ◆ Different techniques are suitable for different stages of testing. Typical white-box testing takes place before behavioral (black-box) testing.
 - ◆ Testing is done by the SW developer and (especially for larger projects) an independent test team.
 - ◆ Testing and debugging are different and separate activities. Debugging must also be included in the test strategy.

Independent Test Group

- ❖ Typically, a developer wants to demonstrate the quality of their product through tests (absence of defects, compliance with requirements ...).
- ❖ The purpose of testing is to look for defects and not to demonstrate quality.
- ❖ The developer is not the preferred test person
 - ◆ There is a conflict of interest
 - ◆ A detailed knowledge of the system means a focused look at the system that is significantly different from the user's thinking models.
- ❖ The independent testing group does not have these limitations
 - ◆ However, the developer needs to work with the testing group to assist in setting up the environment and correcting any errors found.

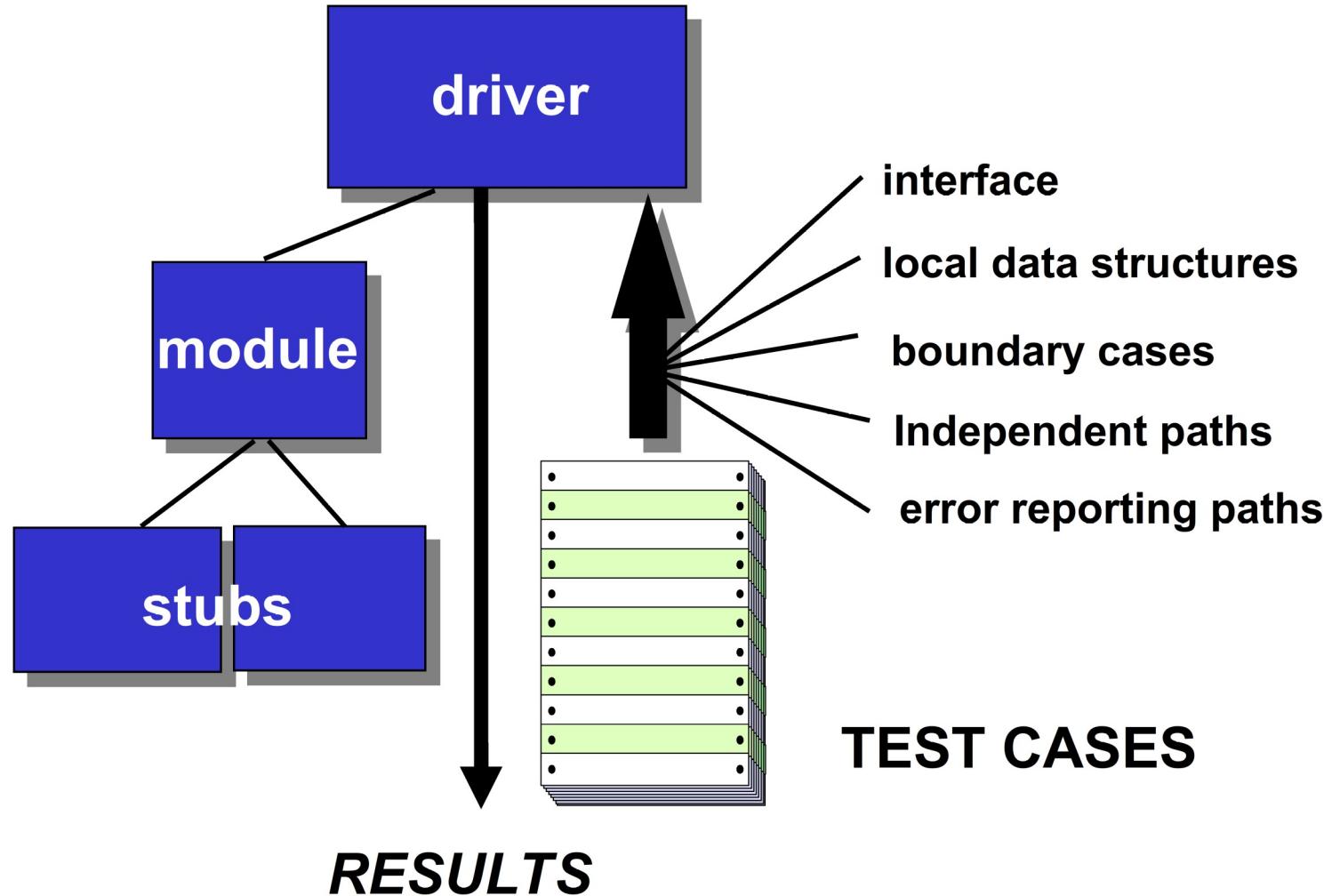
Testing strategies



Unit testing

- ❖ It covers the testing of individual components or units.
- ❖ Testing is based on source code (structural testing / white-box)
- ❖ Typically, a unit test is the responsibility of the developer.
- ❖ The performance of the test depends on the experience of the developer.
- ❖ To test the unit, prepare a test driver.
- ❖ If the unit uses other units, they should be replaced as much as possible by stubs (minimum replacement implementation of the unit).
- ❖ A variety of frameworks are available for unit testing, see:
http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

Unit testing environment

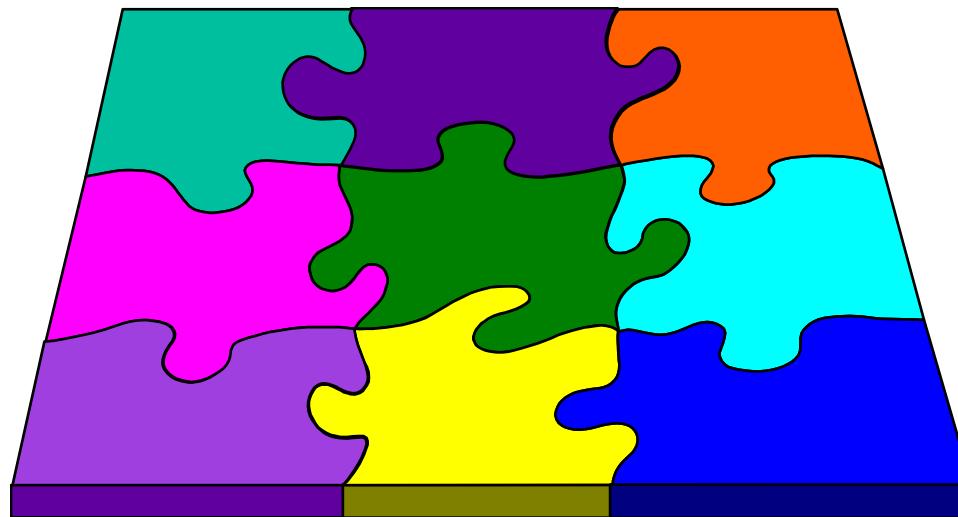


Integration testing

- ❖ Integration testing is intended to test a group of components that are integrated into a system or subsystem.
- ❖ The responsibility of the developers or (better) independent test team.
- ❖ Testing is based on system specifications (requirements, system design) - behavioral testing (black-box).
- ❖ The main problem is error localization, which can be solved by incremental integration.

Integration testing strategies

- ❖ The “big bang” approach
- ❖ Incremental construction strategy



Incremental integration

❖ Top-down integration

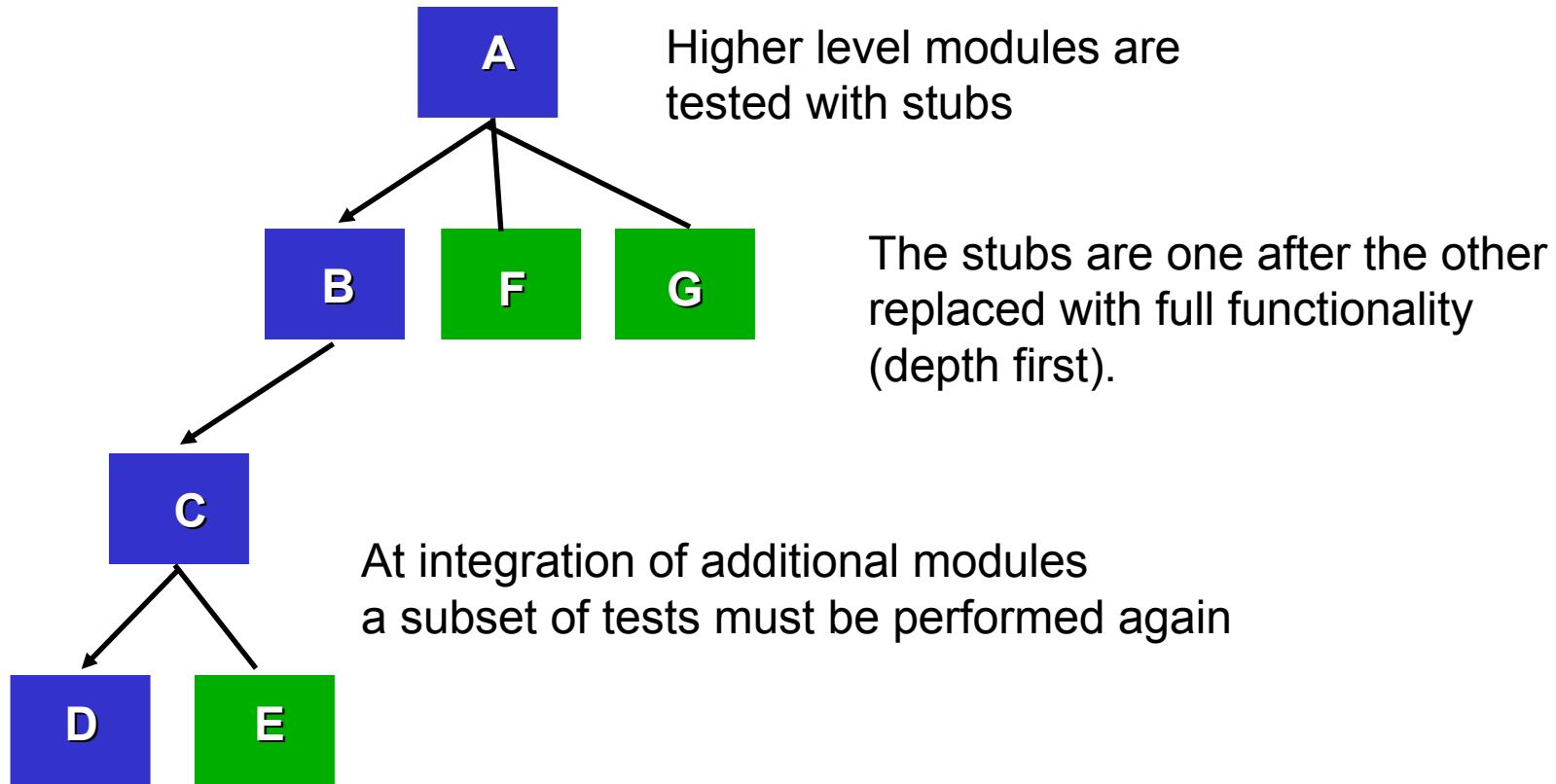
- ◆ First, the high-level components, which are connected to the low-level component stubs.
- ◆ The stubs have the same interface but very limited functionality.

❖ Bottom-up integration (XP)

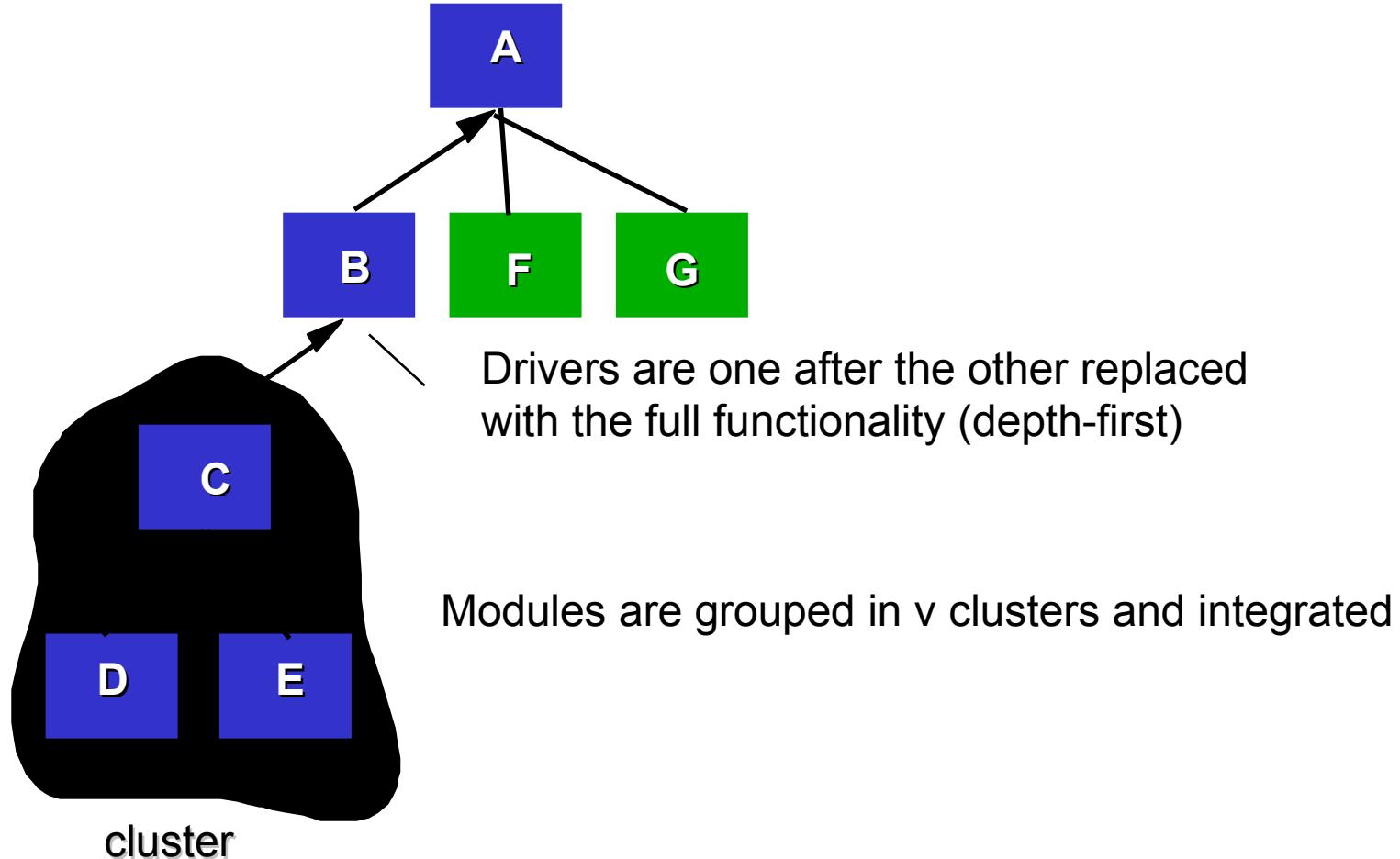
- ◆ First the integration of low-level components with full functionality, then the development and integration of higher-level components until the system is complete.

❖ In practice, it is often a combination of both approaches.

Top-down integration



Bottom-up integration



Testing of interfaces

- ❖ Used when subsystems are integrated into larger systems.
- ❖ The purpose is to find the errors of the interfaces and the errors in the assumptions related to the interfaces.
- ❖ Useful (also) in object-oriented approach, because objects are defined by their interfaces.

Validation testing

- ❖ Validation testing is intended to determine whether a software product is performing in a manner that meets the "reasonable" expectations of users.
- ❖ What is "reasonable" is determined by the requirements specifications.
- ❖ There are two possible results:
 - ◆ Properties meet the requirements
 - ◆ Deviations found and documented in a documented in the deviation list. Deviations often need to be negotiated with the customer (eg due to time constraints).

Validation testing

- ❖ An important part of validation testing is configuration review.
- ❖ This ensures that all elements of the software product are properly developed, cataloged and fit for the maintenance phase.

Validation testing

❖ Alpha and beta testing

- ◆ The developer / tester is often unable to know in detail how the software product is used by the user, so it is appropriate to include end users in the final validation.
- ◆ In the case of individual contracting authorities, the final validation shall be in the form of acceptance tests. These may be informal (test-drive) or formal with actual systematic test cases.
- ◆ When there are many customers, the final validation takes the form of alpha and beta testing.
- ◆ Alpha testing is performed on the developer's side in a controlled environment in the presence of users.
- ◆ Beta testing is performed by one or more clients independently and without the presence of the developer.

System testing

- ❖ System testing is intended to determine the behavior of a software product in the presence of other system elements (hardware, people, information)
- ❖ System testing involves a number of different tests designed to "fully test" the computer system. Different tests identify different properties of the system.
 - ◆ Recovery testing - restoring normal operation after the presence of faults.
 - ◆ Security testing - checking the mechanisms of protection against unwanted interventions.
 - ◆ Stress testing - operation of the system under excessive loads.
 - ◆ Performance testing - system throughput testing.

Stress test

- ❖ Testing the system at a load greater than the maximum planned.
- ❖ It often shows undetected errors.
- ❖ Excessive loading causes errors
 - ◆ Which should not be catastrophic.
 - ◆ There should be no tampering or loss of data
- ❖ Particularly important in distributed systems.

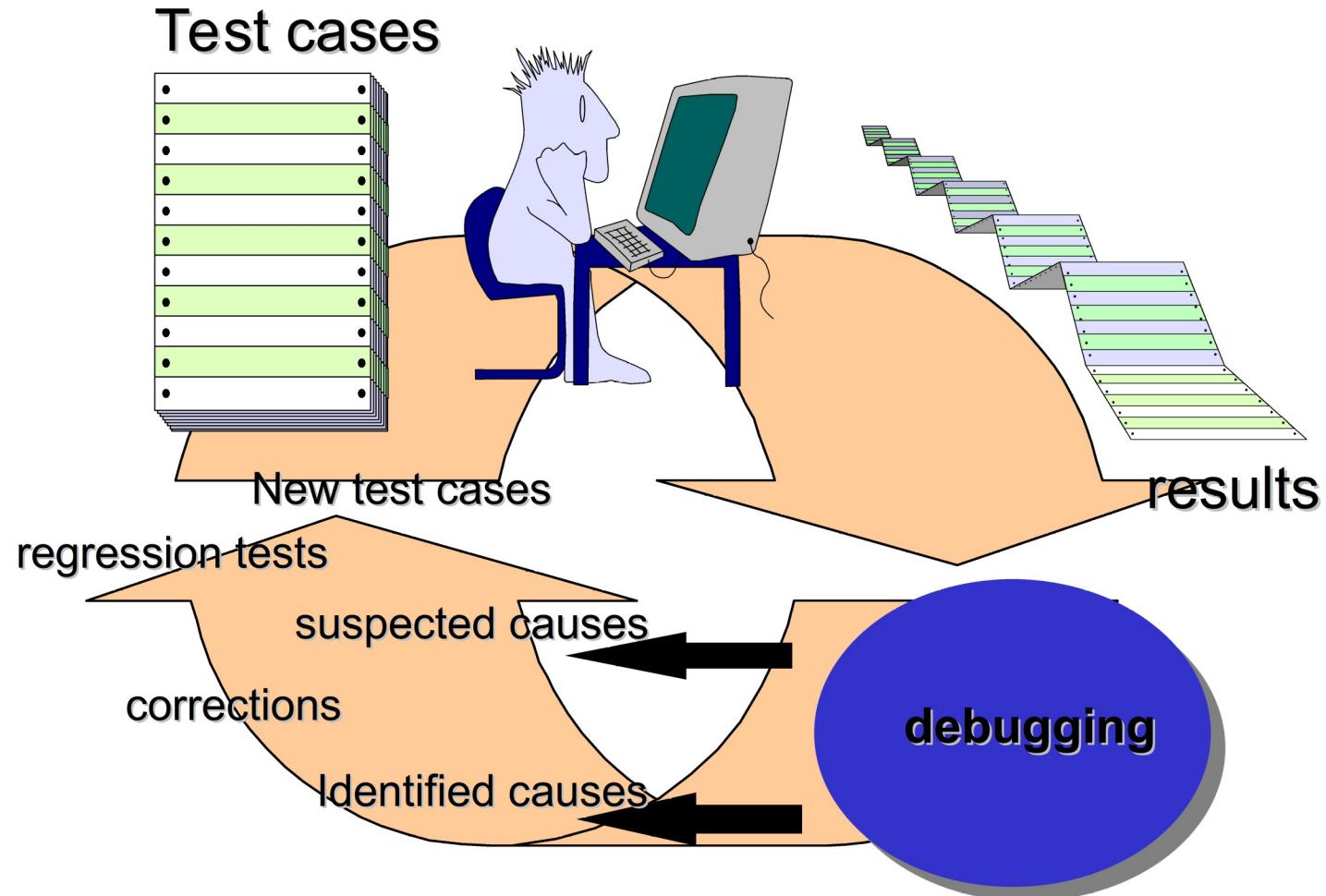
Debugging

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

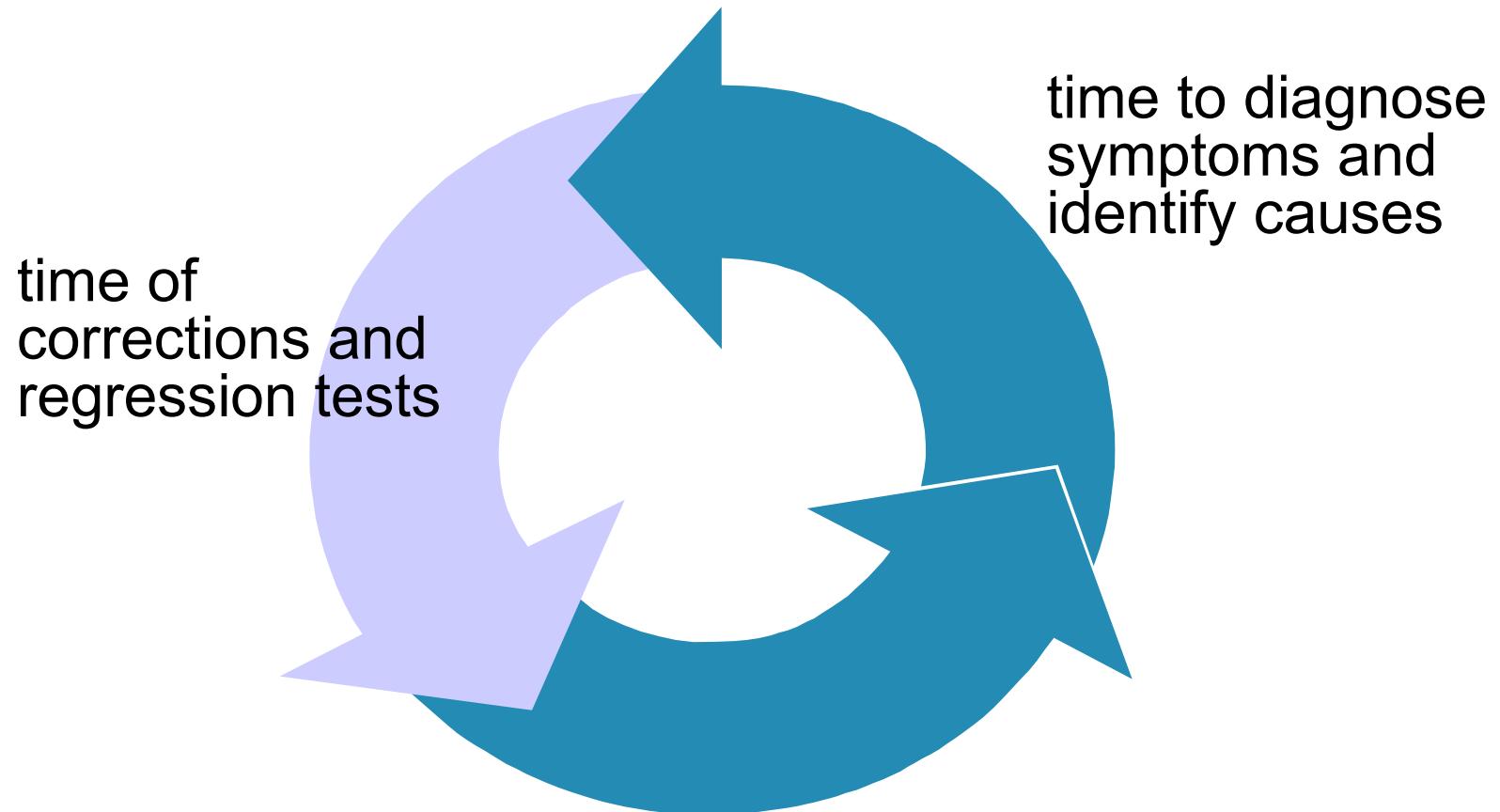
Debugging

- ❖ Debugging is a diagnostic process as a result of successful testing.
- ❖ The purpose is to correct the errors found.
- ❖ The manifestation of the error (symptom) and the cause of it do not necessarily have an obvious interconnection.
- ❖ Debugging is a thought process that connects a symptom to the cause of an error.

Debugging process

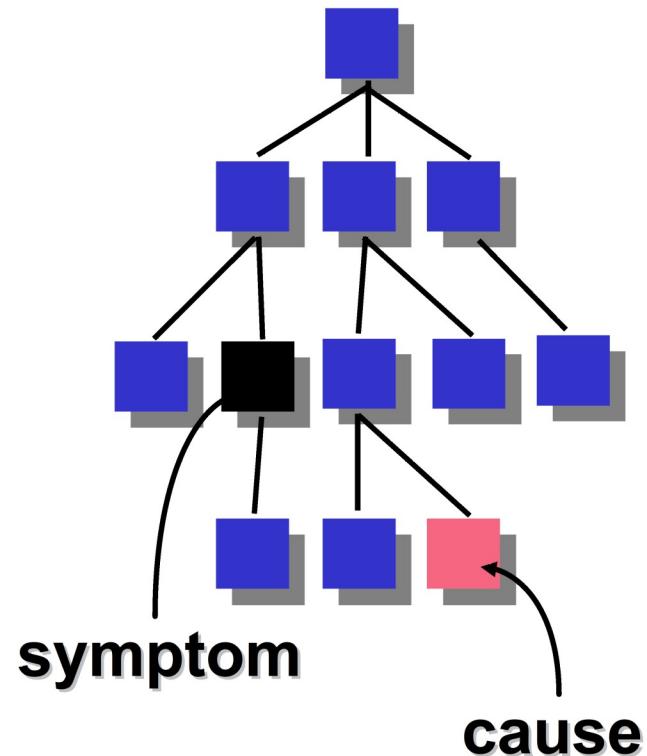


Debugging effort

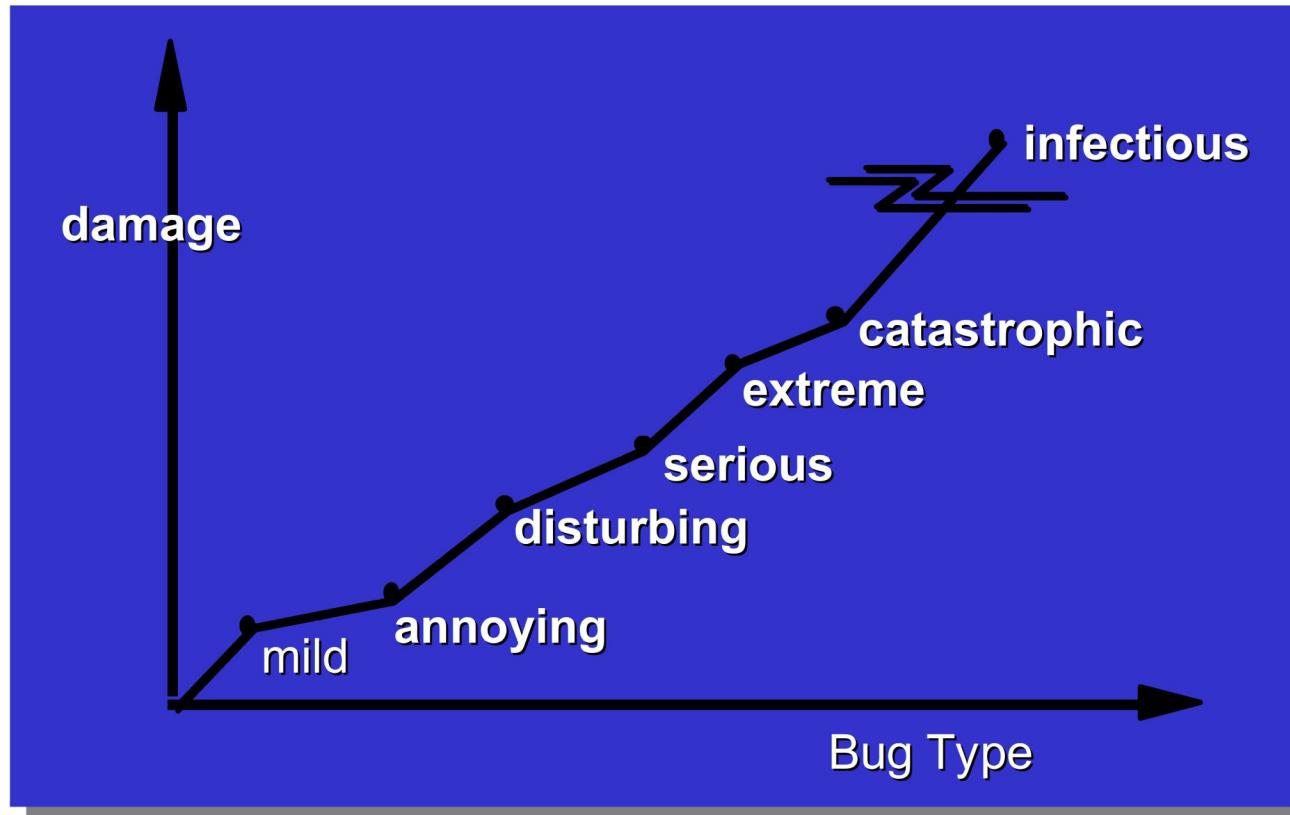


Causes and symptoms

- ❖ Symptoms and causes of errors may be geographically distinct.
- ❖ The symptom may disappear due to the elimination of another error.
- ❖ The reason for the symptom may be a combination of non-errors.
- ❖ The error may be due to a system error or a code compilation.
- ❖ The reason may lie in the assumptions that everyone believes.
- ❖ The symptom may be volatile.



Bug consequences



Bug Categories: function-related bugs, system-related bugs, data bugs, coding bugs, design bugs, documentation bugs, standards violations, etc.

Debugging

- ❖ Techniques
 - ◆ Extensive testing
 - ◆ Tracking
 - ◆ Induction
 - ◆ Deduction
- ❖ Use tools (eg debugger) to get a better idea of what's happening in the SW
- ❖ After "eliminating" the error, remember to perform regression testing!

Documenting

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Documenting

- ❖ IEEE 829 is a Standard for Software Test Documentation and defines a set of recommendations for documenting software product testing.
- ❖ IEEE 829 identifies eight different documents covering three testing steps:
 - ◆ Preparation for testing
 - ◆ Testing execution
 - ◆ Completion of testing
- ❖ IEEE 829 is a good starting point for determining test documents. However, the documents must be adjusted/modified to meet the needs of the project.
- ❖ Use your documents to meet your needs and not for your own purpose!

IEEE 829 - Preparation for testing

- ❖ **Test plan**
 - ◆ It defines how the test will be conducted: what will be tested, who will test, how will it be tested and when will it be tested.
- ❖ **Test design specification**
 - ◆ A detail of the test conditions and the expected outcome. This document also includes details of how a successful test will be recognized.
- ❖ **Test case specification**
 - ◆ A detail of the specific data that is necessary to run tests based on the conditions identified in the previous stage.
- ❖ **Test procedure specification**
 - ◆ A detail of how the tester will physically run the test, the physical set-up required, and the procedure steps that need to be followed.
- ❖ **Test item transmittal report**
 - ◆ It states what has been submitted to the testing process.

IEEE 829 – Testing execution

- ❖ **Test log**
 - ◆ A detail of what tests cases were run, who ran the tests, in what order they were run, and whether or not individual tests were passed or failed.
- ❖ **Test incident report**
 - ◆ A detail of the actual versus expected results of a test, when a test has failed, and anything indicating why the test failed.

IEEE 829 - Zaključek testiranja

❖ Test summary report

- ◆ A detail of all the important information to come out of the testing procedure, including an assessment of how well the testing was performed, an assessment of the quality of the system, any incidents that occurred, and a record of what testing was done and how long it took to be used in future test planning. This final document is used to determine if the software being tested is viable enough to proceed to the next stage of development.

Test plan (IEEE 829)

- 1) Test Plan Identifier
- 2) References
- 3) Introduction
- 4) Test Items
- 5) Software Risk Issues
- 6) Features to be Tested
- 7) Features not to be Tested
- 8) Approach
- 9) Item Pass/Fail Criteria
- 10) Suspension Criteria and Resumption Requirements
- 11) Test Deliverables
- 12) Remaining Test Tasks
- 13) Environmental Needs
- 14) Staffing and Training Needs
- 15) Responsibilities 16) Schedule
- 17) Planning Risks and Contingencies
- 18) Approvals
- 19) Glossary

See: <https://jmpovedar.files.wordpress.com/2014/03/ieee-829.pdf>

Test plan (general)

- ❖ A test plan is a document that defines a complete set of system test cases and additional information about the test process.
 - ◆ The absence of a test plan means ad-hoc testing, which often results in poor quality software products.
 - ◆ The test plan should be prepared well in advance of testing.
 - ◆ The preparation of the test plan may commence as soon as the specification of the requirements is completed.

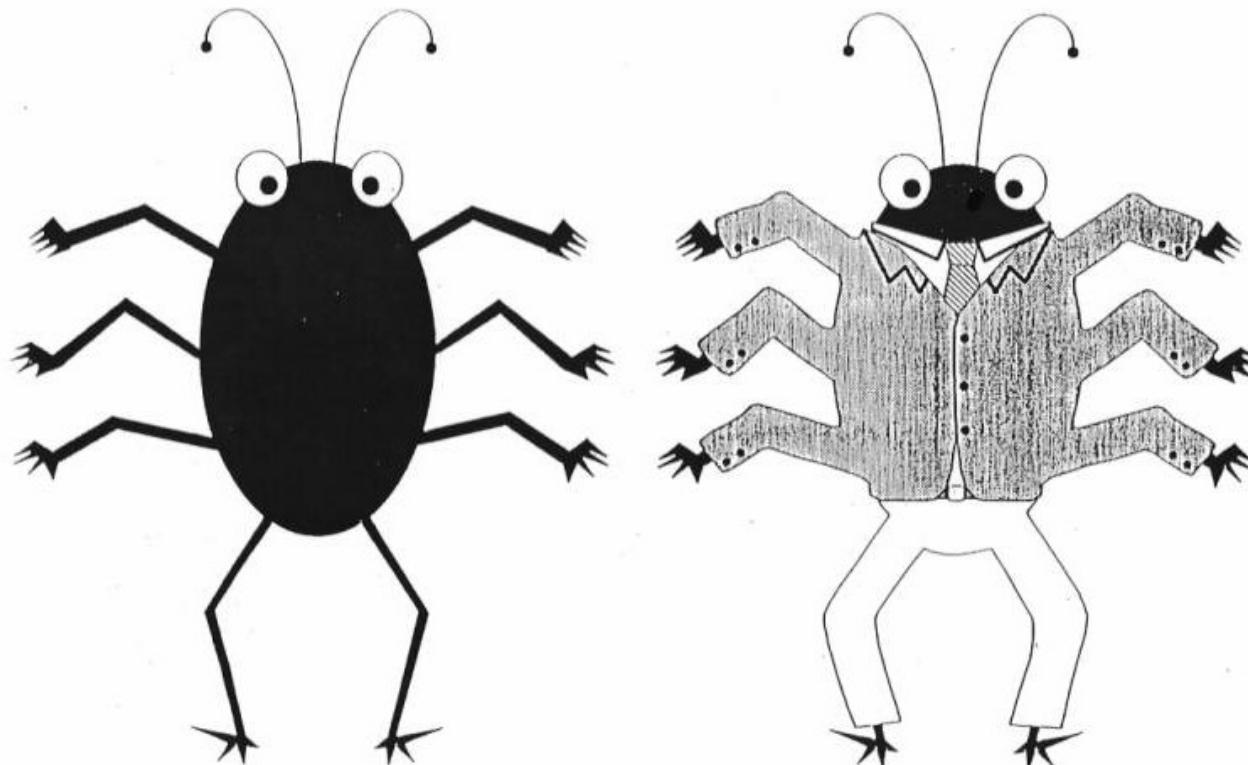
Test cases

- ❖ The test case is an explicit set of instructions for detecting specific errors in the software system.
 - ◆ The test case is used for a large number of tests.
 - ◆ Each test is the execution of a specific test case on a specific version of the system.
- ❖ The test case documentation contains:
 - ◆ Identifier and classification:
 - Test case number and descriptive name.
 - Indication of the system, subsystem or module to which it relates.
 - Indication of the importance of the test.
 - ◆ Directions:
 - Accurate instructions to the examiner for what to do. Typically he does not refer to other documents.
 - ◆ Expected results:
 - Description of what the system is supposed to do as a result of running a test case
 - The tester reports an error if the system response does not match the expected one.
 - ◆ Cleaning (if necessary):
 - Tells the tester how to bring the system back to 'normal' after the test.

Reports

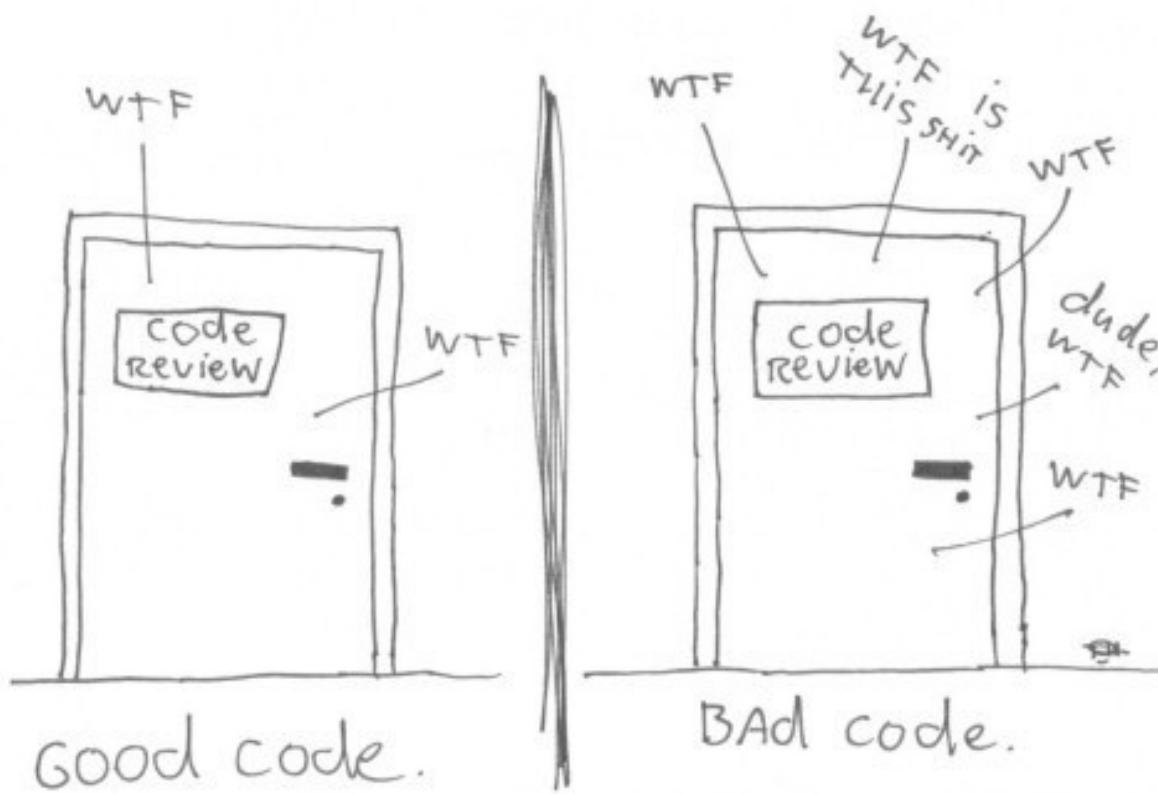
- ❖ Test results may depend on the environment, so test reports should include an identification of the environment (operating systems, versions of databases, and related systems ...).
- ❖ The results may depend on other SW running concurrently with the system tested!
- ❖ Example test reports:

<https://strongqa.com/qa-portal/testing-docs-templates/test-report>



BUG **FEATURE**

The ONLY VALID measurement OF Code QUALITY: WTFs/minute



Software engineering

Maintenance

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Deployment and maintenance

❖ Deployment

- ◆ Upon acceptance of the SW product, the customer may perform an acceptance test. This can be done
 - at the developer (factory acceptance test - FAT) or
 - at the customer (site acceptance test - SAT).
- ◆ The whole SW system is delivered to the customer (executable SW, documentation and data required for operation). This is followed by installation and transition to operation.

❖ Operation and maintenance

- ◆ During operation the user documentation must be available, in addition user training and user support may be provided.
- ◆ The developer maintains software products during the operation.
- ◆ Configuration management is important for the effective maintenance of software products.

Documentation and training

- ❖ The documentation is intended for two target groups:
 - ◆ System maintainers
 - ◆ System users who (daily) use the system.
- ❖ User training
 - ◆ Can be adapted to the specific needs of users or
 - ◆ general (in the case of consumer products).

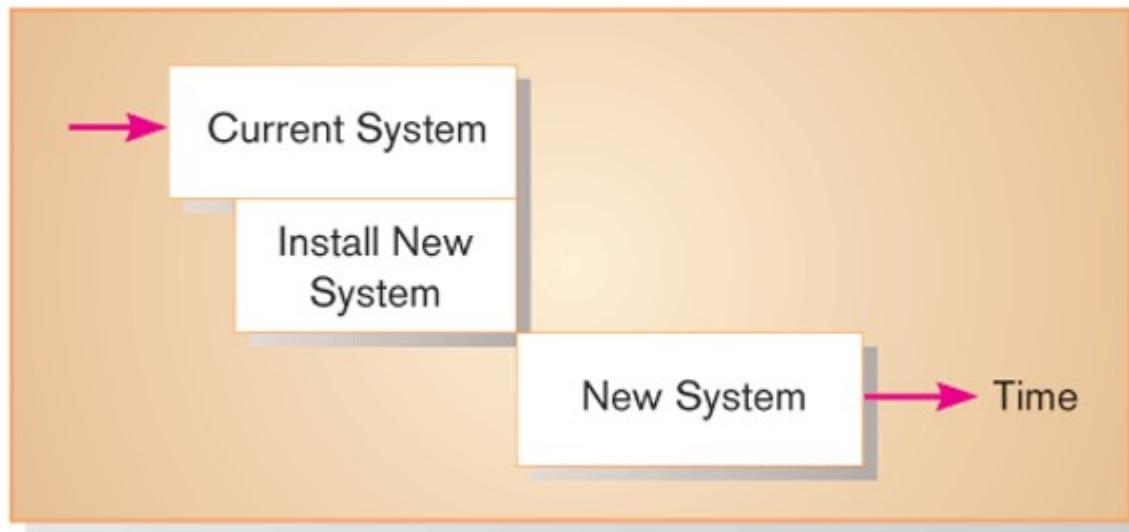
Customer support

- ❖ Customer support is extremely important for users.
- ❖ Providing customer support can be expensive (customer support service must be established).

Installation

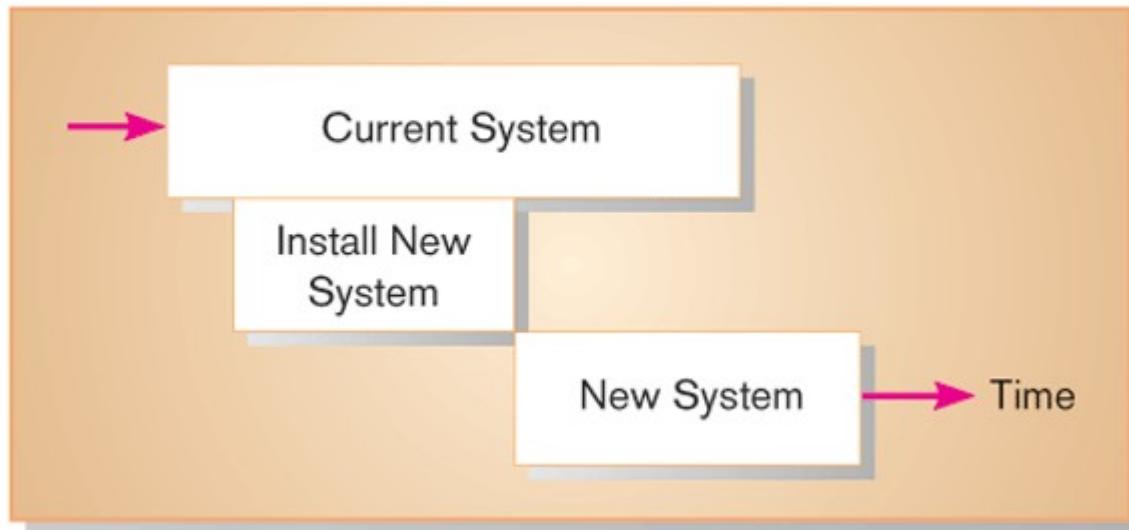
- ❖ System installation is an organizational process in which an existing system is replaced by a new one.
- ❖ There are four installation strategies:
 - ◆ Direct installation
 - ◆ Parallel installation
 - ◆ Installation at a specific location
 - ◆ Phase installation

Direct installation



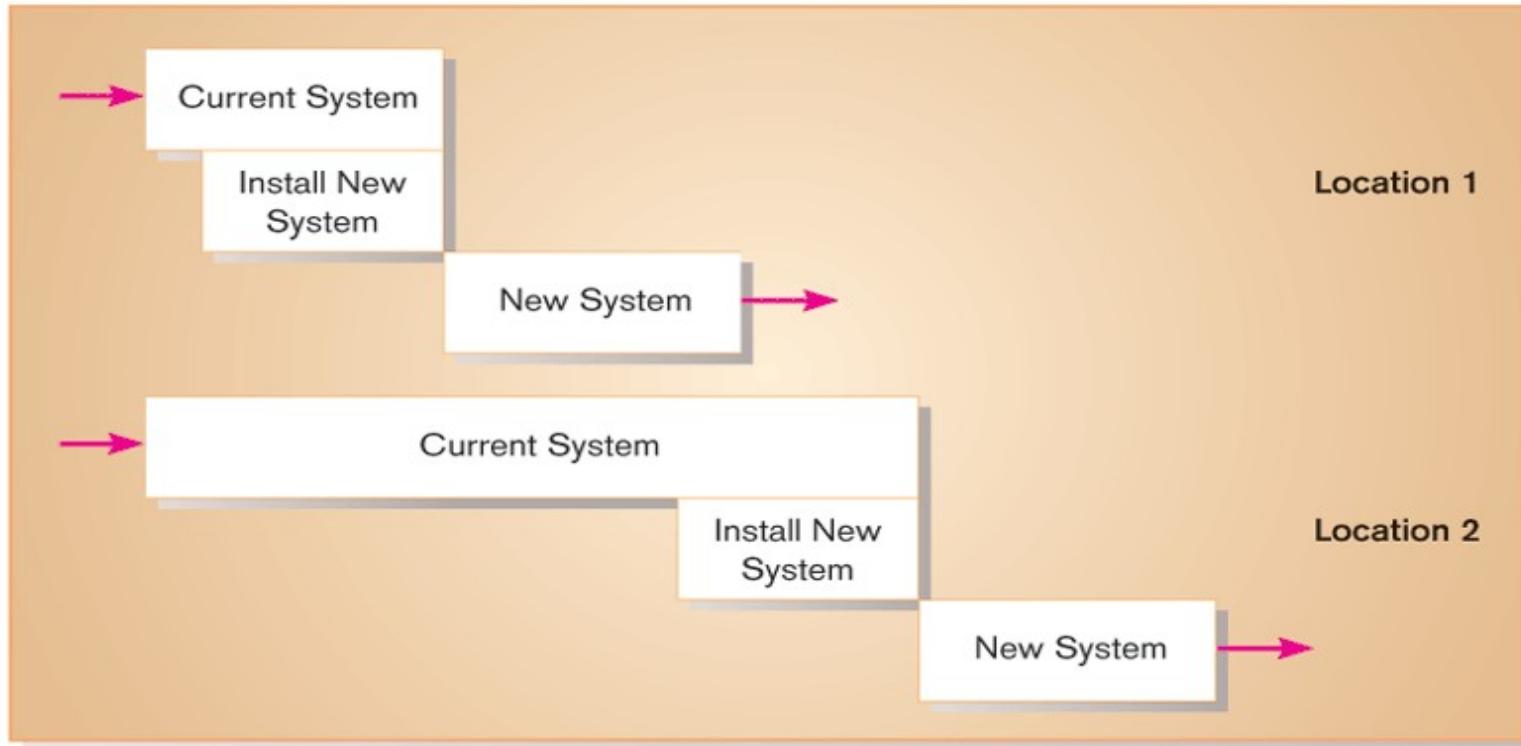
- ❖ When the new system starts, the old system stops.

Parallel installation



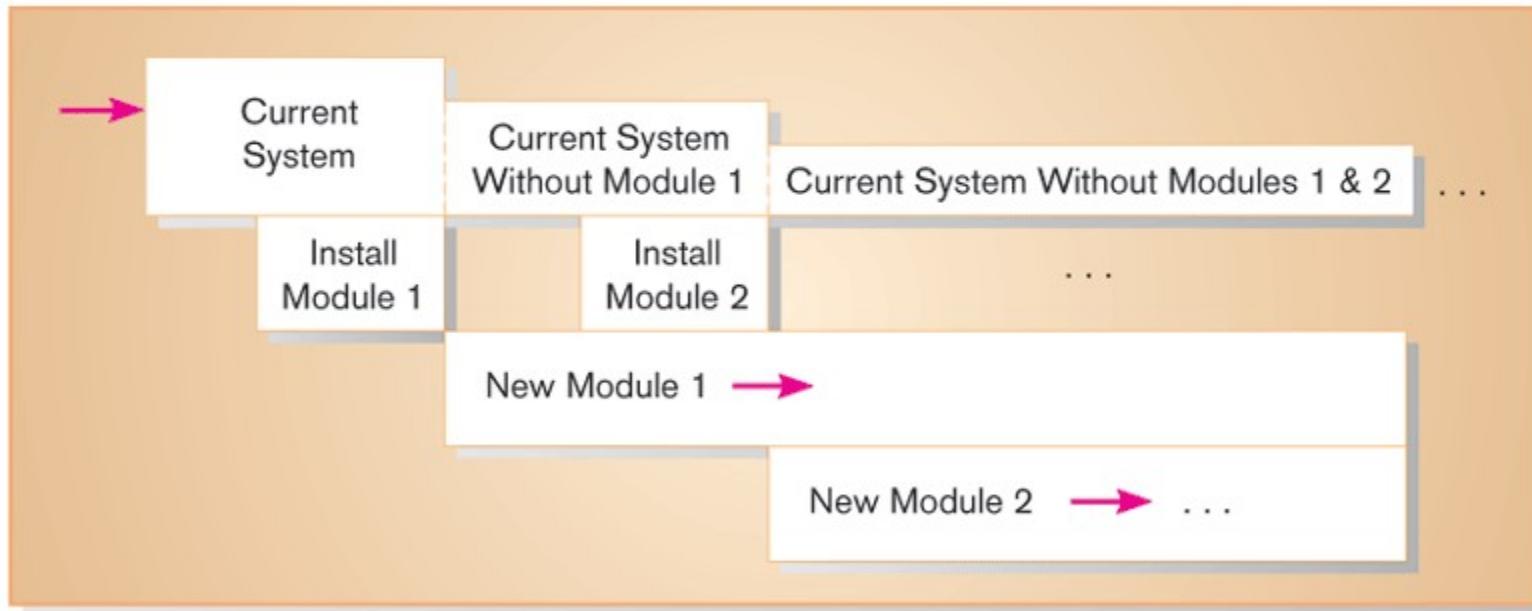
- ❖ The old and the new system work in parallel until management decides to stop the old system.

Installation at a specific location



- ❖ The new system is installed in a single starting location. It is up to the site to decide if and how the system will expand to other locations throughout the organization.

Phase installation



- ❖ The transition from the old to the new system is gradual. It begins with the replacement of one (or several) functional components, and then the components are gradually replaced until the new system is fully installed.

Installation planning

What needs to be considered:

- ❖ Data conversion of existing system
 - ❖ Data error correction
 - ❖ Transferring data from an existing system
- ❖ Planning to stop the existing system and switch to the new system.
- ❖ Consideration of switching in the business process of an organization.

Success factors

- ❖ The most important measure of the success of a SW product and its development is the extent of its usage.
- ❖ The use of the system is influenced by a number of factors:
 - ◆ Personal interest of users
 - ◆ System features
 - ◆ Demographics (gender, age, race, income, education, etc.)
 - ◆ Organizational support
 - ◆ System performance
 - ◆ User satisfaction

Finalizing the project

- ❖ Evaluation of the project (reviews).
- ❖ Evaluation of the project team
 - ◆ Assignment of members to other projects
- ❖ Informing all participants of project completion and transition to operation and maintenance.
- ❖ Completion (redemption) of contracts with clients.
- ❖ Formal closure of the project

Maintenance

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Maintenance

❖ Maintenance in general:

- ◆ Functional checks, servicing, repairing or replacing of components and all other activities that keep the product in operation.

❖ Software products:

- ◆ Maintenance? Software products do not wear out or break down and do not require periodic maintenance in that manner.
- ◆ Maintenance means changing a software product after release for the purpose of:
 - error corrections,
 - performance improvements (system throughput ...)
 - adaptation of the product to the changed environment.
 - adapting to new requirements

Maintenance is crucial for maintaining the use of a (software) product!

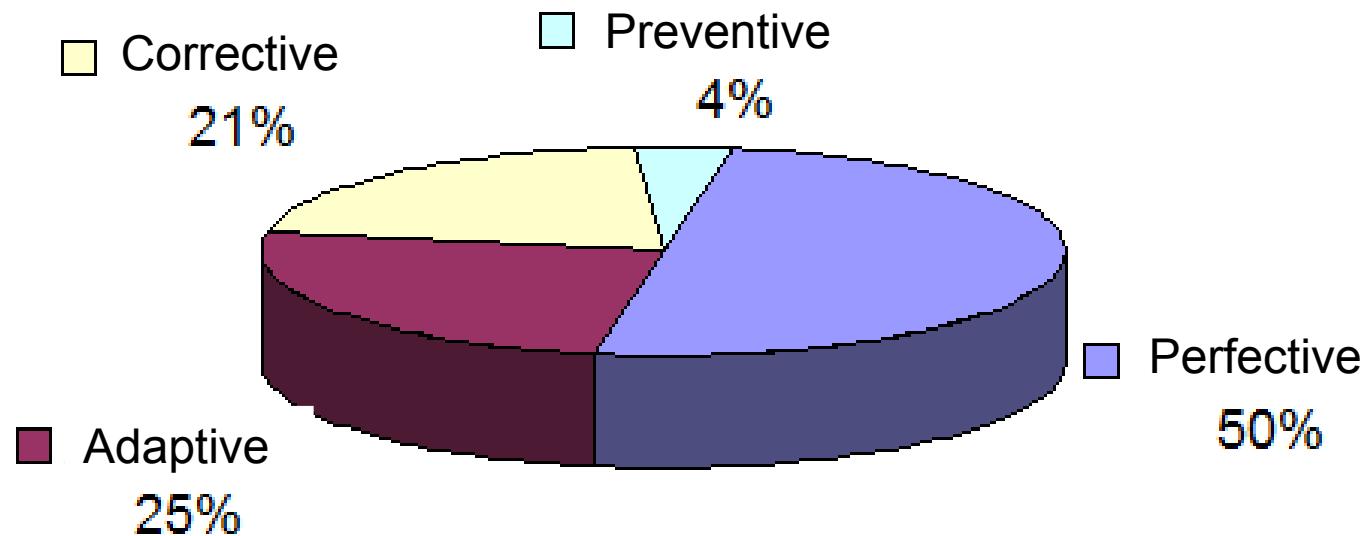
Maintenance cost

- ❖ Maintenance is expensive!
Typically, the cost of maintenance is 2 to 100 times higher than the cost of development!
- ❖ The cost (difficulty) of maintenance is influenced by both technical and non-technical factors.
 - application type, system uniqueness, personnel fluctuation and experience, system lifetime, variable environment dependency, hardware characteristics, design quality (structure), source code quality, documentation quality, testing quality, number of users, use of tools ...

Changes of software products

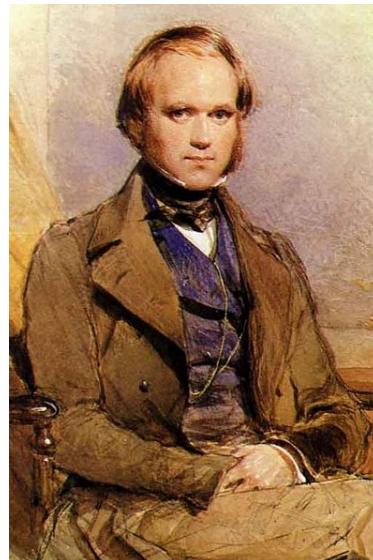
- ❖ In most cases, the starting point for maintenance is the existing product (this is inconsistent with the linear development model).
- ❖ Maintenance includes 4 types of software changes:
 - ◆ Corrective changes
 - Error correction.
 - ◆ Adaptive changes
 - Adaptation to changed conditions.
 - ◆ Perfective changes
 - Improvements of the software product.
 - ◆ Preventive changes
 - Elimination of software product degradation.

Percentage of change types

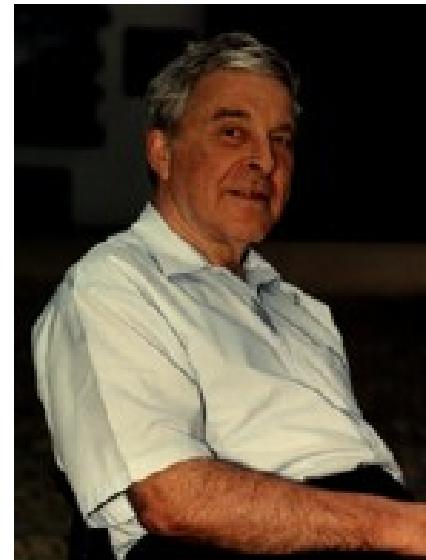


Evolution

- ❖ Software maintenance is software evolution!
- ❖ **Evolution:** the gradual change of something, usually into more complete, more sophisticated forms, development (source: SSKJ).

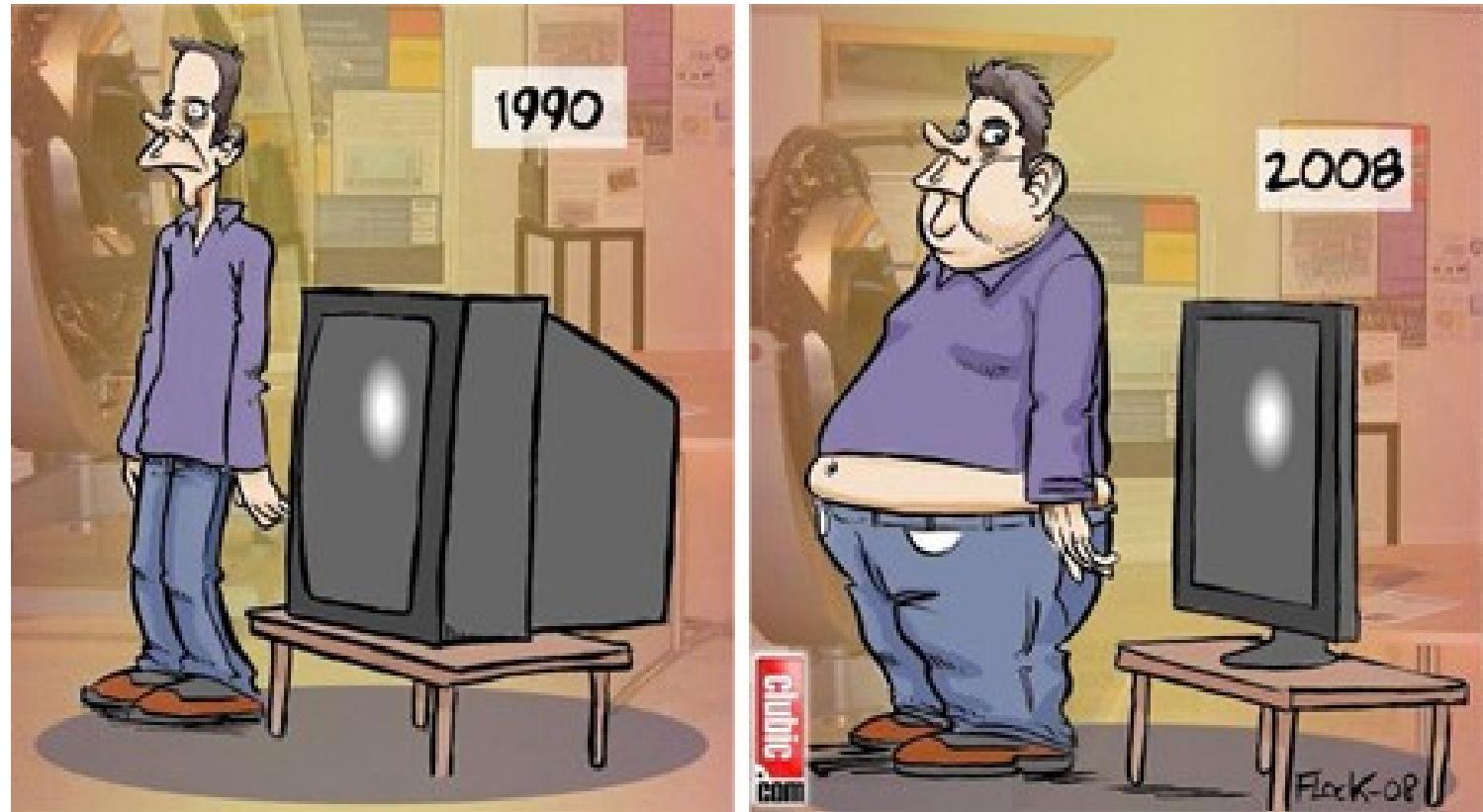


Charles Darwin



Meir Manny Lehman

Evolution ☺



What is evolving?

- ❖ Requirements specification
 - ◆ Requirement evolution
- ❖ Design
 - ◆ Architecture evolution
- ❖ Implementation
 - ◆ Data evolution
 - ◆ Source code evolution
 - ◆ Documentation evolution
 - ◆ Evolution of technology used
- ❖ Testing
 - ◆ Testing evolution (test cases)

Evolution must be synchronous: co-evolution problem

Evolutionary types of SW prod.

- ❖ Meir Manny Lehman identified three types of software products according to the need for maintenance.
 - ◆ S-system: Formally defined systems that do not require change throughout their lifespan. They rarely occur in practice!
 - ◆ P-system: systems whose requirements are based on an applied solution to a problem. They need incremental adaptation to the real world.
 - ◆ E-system: systems embedded in the real world require constant adaptation to the real world. Most software products belong to this group!

Lehman's laws of SW evolution

I	Continuing Change	An E-type system must be continually adapted else it becomes progressively less satisfactory in use
II	Increasing Complexity	As an E-type system is evolved its complexity increases unless work is done to maintain or reduce it
III	Self regulation	Global E-type system evolution processes are self-regulating
IV	Conservation of Organisational Stability	Average activity rate in an E-type process tends to remain constant over system lifetime or segments of that lifetime
V	Conservation of Familiarity	In general, the average incremental growth (growth rate trend) of E-type systems tends to decline
VI	Continuing Growth	The functional capability of E-type systems must be continually enhanced to maintain user satisfaction over system lifetime
VII	Declining Quality	Unless rigorously adapted to take into account changes in the operational environment, the quality of an E-type system will appear to be declining as it is evolved
VIII	Feedback System	E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems

1. Law of continuing change

Systems must be continuously adapted or they become progressively less satisfactory to use. The variance between the system and its operational context leads to feedback pressure forcing change in the system

6. Law of continuous growth

Functional capability must be continually increased over a system's lifetime to maintain user satisfaction. In any system implementation, requirements have to be constrained. Attributes will be omitted, these will become the irritants that trigger future demand for change.
Feedback from the users.

2. Law of increasing complexity

As a system evolves, its complexity increases unless work is done to maintain or reduce it. If changes are made with no thought to system structure, complexity will increase and make future change harder. On the other hand, if resource is expended on work to combat complexity, less is available to system change. No matter how is balance is reconciled, the rate of system growth inevitably slows.

7. Law of declining quality

Unless rigorously adapted to meet changes in the operational environment, system quality will appear to decline. A system is built on a set of assumptions, and however valid these are at the time, the changing world will tend to invalidate them. Unless steps are taken to identify and rectify this, system quality will appear to decline, especially in relation to alternative products that will come onto the market based on more recently formulated assumptions.

Validity of Lehman's laws

- ❖ Lehman's laws fit well with commercial software.
- ❖ In the case of "non-standard" software, the Lehman's laws may not be valid.
 - ◆ Open source software...
 - Linux example...
 - ◆ Different development basis:
 - developers are solving their own problems,
 - less time pressure,
 - ...

Maintenance difficulty

- ❖ The gap between:
 - ◆ Need for change (error correction, customization)
 - ◆ The need for the availability of the system for users (tendencies for minimal and quick implementation of changes, patches).
- ❖ The decision must be made between:
 - ◆ Fast and messy problem solving
 - ◆ In-depth and elegant solutions.

The end of maintenance?

- ❖ With the decision to discontinue maintenance, the software begins to "decay".
- ❖ The decision to evolve or discontinue maintenance is related to the following questions:
 - ◆ Is the maintenance cost too high?
 - ◆ Is the system reliability insufficient?
 - ◆ Is the system no longer able to adapt to changes fast enough?
 - ◆ Is the system performance insufficient?
 - ◆ Are the system's functionalities of limited usefulness?
 - ◆ Can other systems do the same job better, faster, cheaper?
 - ◆ Is the cost of maintaining the hardware high enough to warrant replacement with a new, cheaper one?

Who does the maintenance?

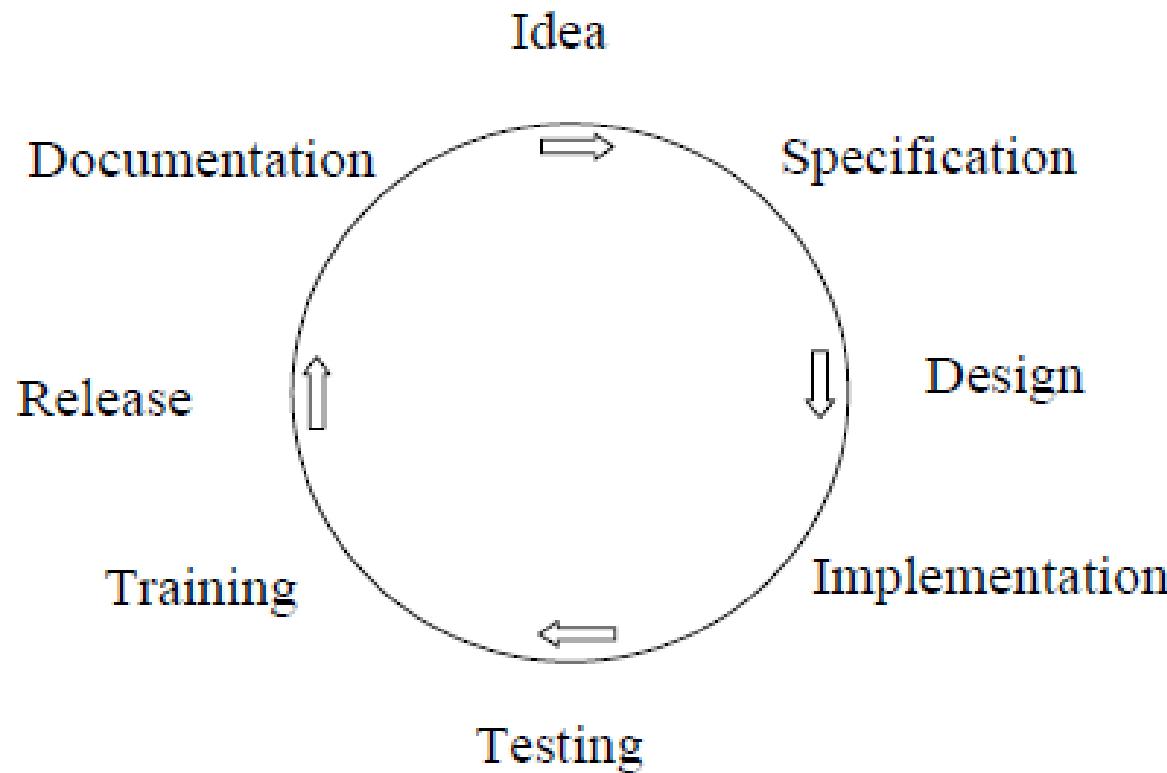
- ❖ A separate maintenance team
 - ◆ It can be more objective
 - ◆ It might be easier to distinguish between how the system should work and how it actually works.
 - ◆ It enables the members of the old team to focus to new projects.
- ❖ Maintenance is performed by a part of the development team
 - The system will be built to make maintenance easier.
 - Team members may feel overconfident and neglect documentation that helps with maintenance.

Maintenance process

The main steps are:

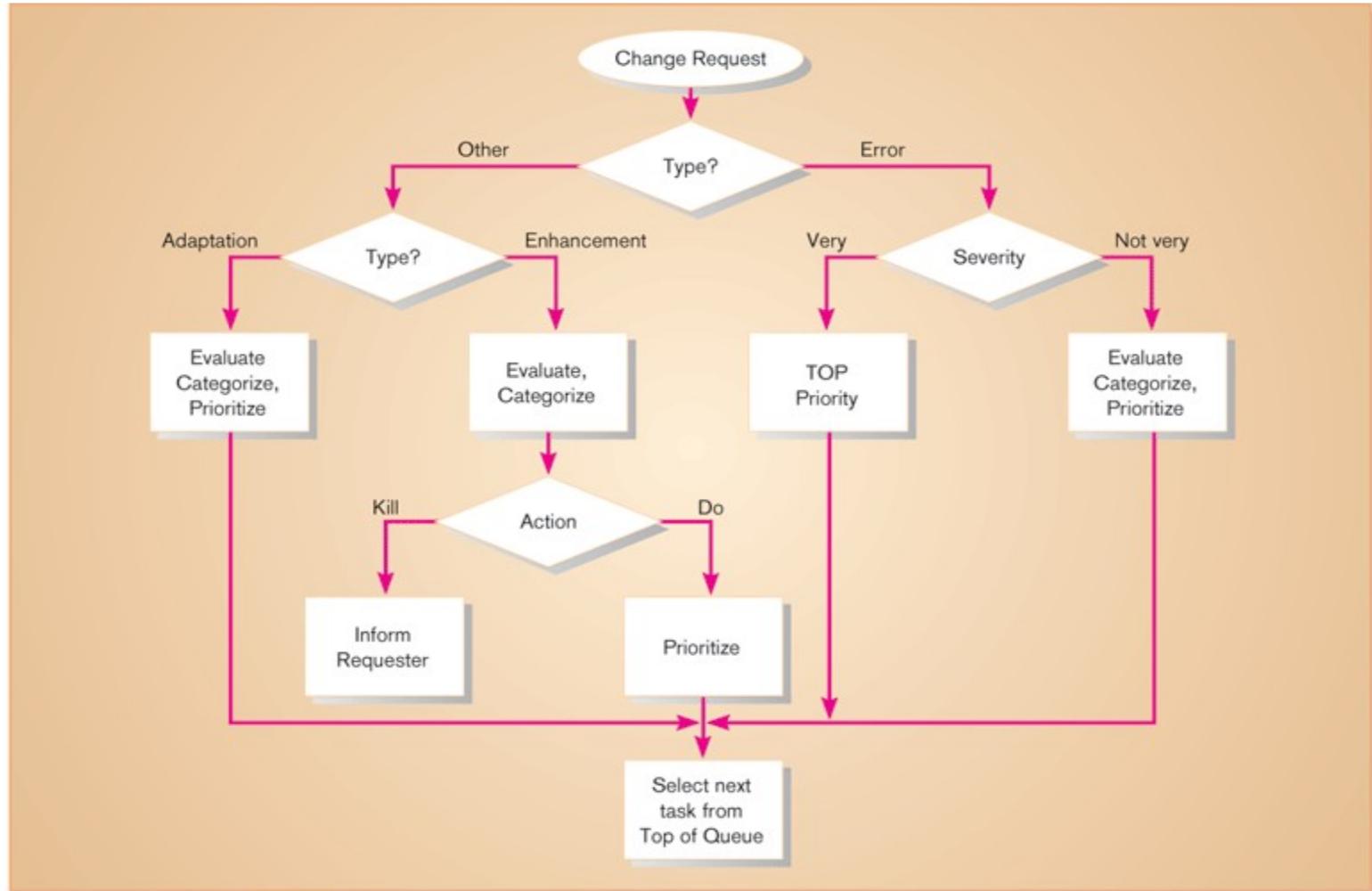
- ◆ Obtaining maintenance applications /requests/ ideas.
- ◆ Specifying a change request.
- ◆ Designing the change
- ◆ Implementing changes
- ◆ Testing
- ◆ Documenting
- ◆ Training
- ◆ Deployment

Maintenance process



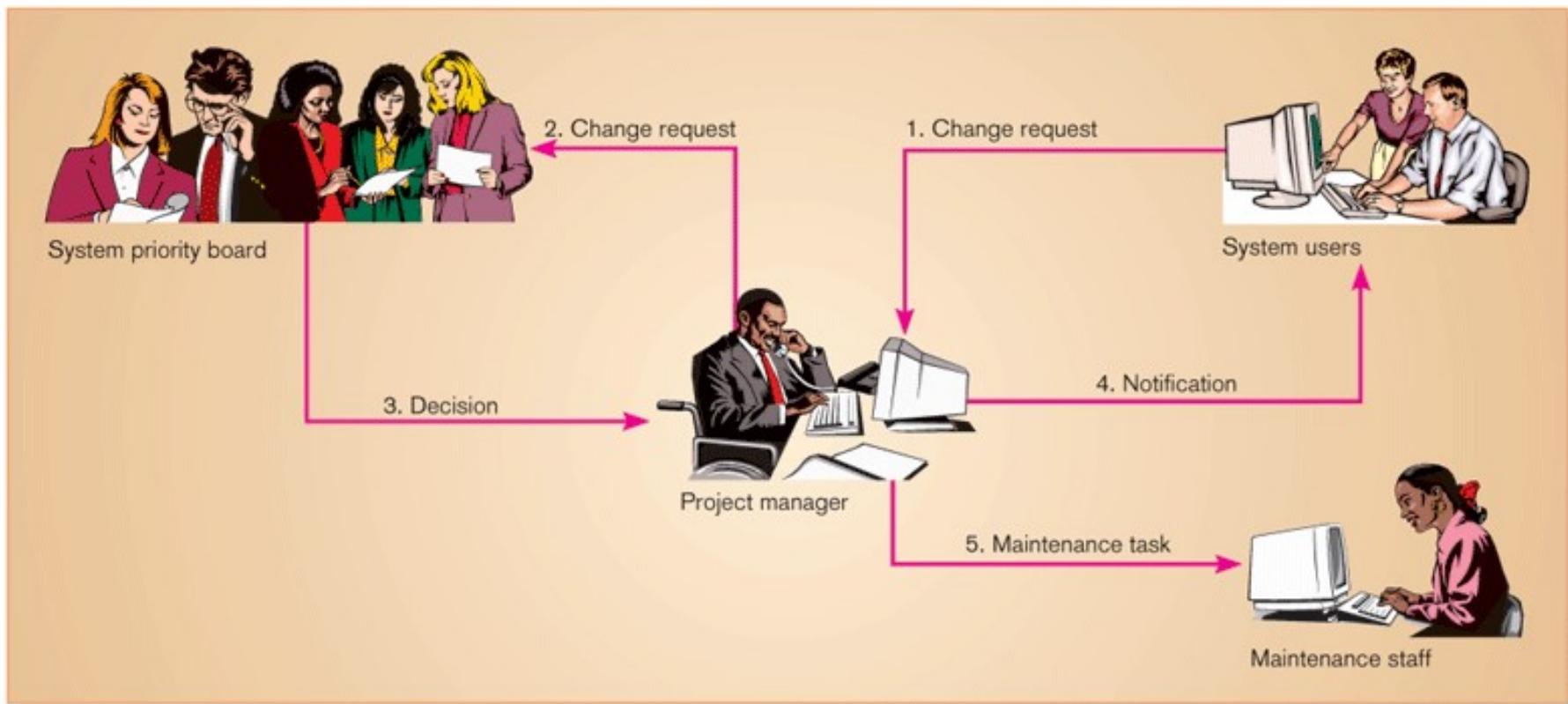
A simplified maintenance process (without returning in previous phases)

Handling of maintenance requests

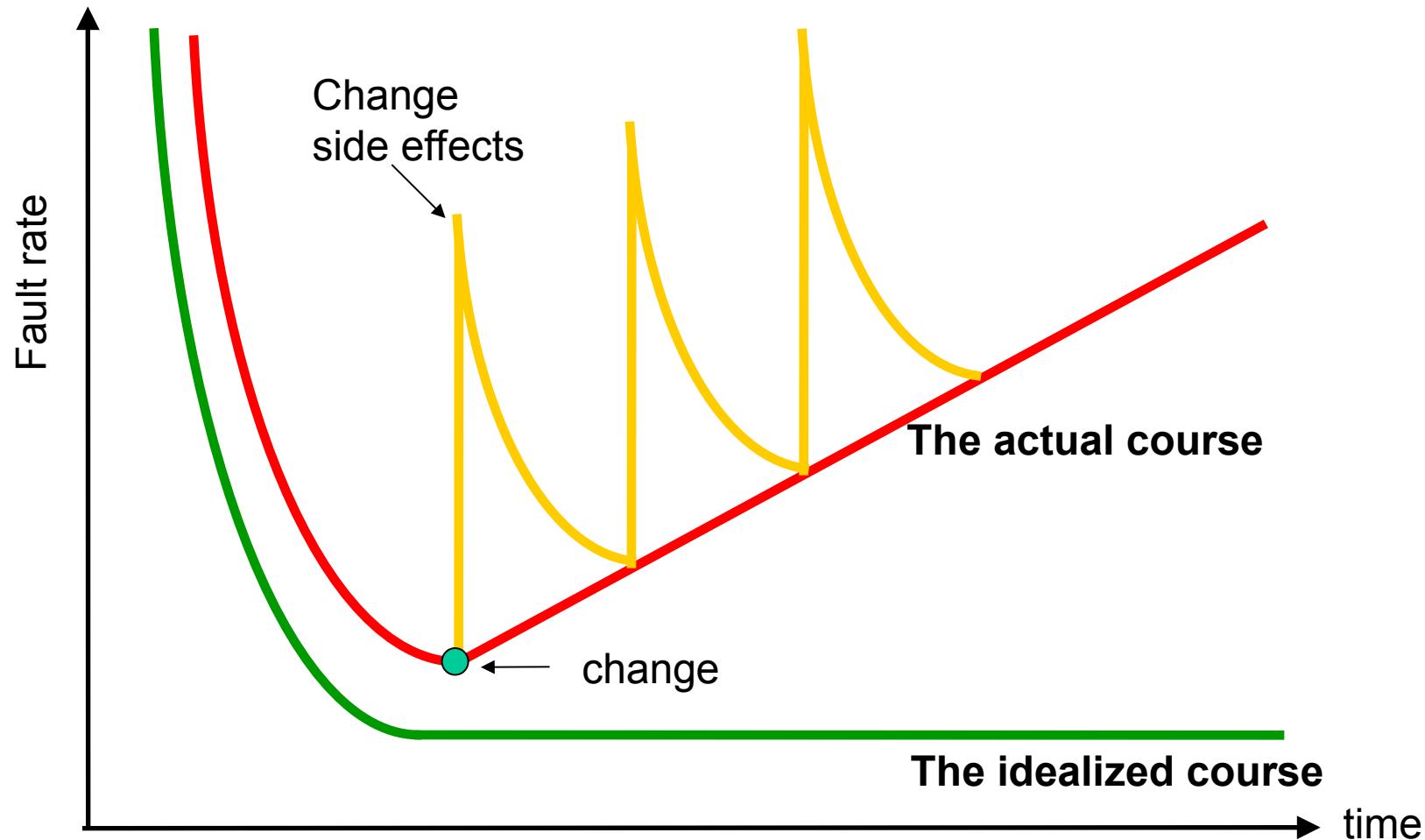


Source: Pressman, R. S. 2001. *Software Engineering: A Practitioner's Approach*, 5th Edition. New York: McGraw-Hill.

Change request path



Remember!



Maintenance techniques and tools

- ❖ Configuration management
- ❖ Impact analysis
- ❖ Engineering, reverse engineering, re-engineering.

Engineering

- ❖ **Forward engineering** is a traditional process that runs from a high level of abstraction and logical, execution-independent design to the physical execution of the system.
- ❖ **Reverse engineering** is the process of analyzing a system to identify system components, their interactions, and represent the system in a different form with a higher degree of abstraction.
- ❖ **Reengineering** is the exploration and conversion of a system to reconstruct and implement it into a new design with the same external functionality.

Rationale

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Rationale?

- ❖ **Rationale:** Explanation of the logical reasons or principles employed in consciously arriving at a decision or estimate. Rationales usually document
 - (1) why a particular choice was made,
 - (2) how the basis of its selection was developed,
 - (3) why and how the particular information or assumptions were relied on, and
 - (4) why the conclusion is deemed credible or realistic. (<http://www.businessdictionary.com/definition/rationale.html>)
- ❖ **Racionale:**
A documented and reasoned decision that determines how a product is developed in its software process.

Rationale

- ❖ Rationale includes
 - ◆ Issues to which it relates
 - ◆ Alternatives considered
 - ◆ Decisions taken to resolve the issue
 - ◆ The criteria that led to the decision and
 - ◆ Discussions that were needed in the development team to reach a decision.

Aviation example

- ❖ **Airbus A320 (1988)**

The first passenger aircraft with a fly-by-wire control system.

150 seats, short / medium range.

- ❖ **Airbus A319, A321**

Both derived from the A320

Same fly-by-wire controls

- ❖ **Design rationale:**

Reduced training and maintenance costs,

Increased flexibility of airlines.



Aviation example (2)

- ❖ Airbus A330, A340

- ❖ Long/very long range (3xA320)
- ❖ Double no. seats like the A320
- ❖ Similar flight controls as A320



- ❖ Design rationale

- ❖ A320 pilots require minimal additional training to fly the A330, A340.

- ❖ Consequences

Any change of the five aircraft types must maintain their similarity (in operation).

Software engineering

Agile methodologies for
SW development

doc. dr. Peter Rogelj (peter.rogelj@upr.si)

Managing uncertainty

- ❖ Software development is a learning process to reduce uncertainty:
 - ◆ Project scope
 - Determine the functionality
 - ◆ Resources required
 - Development team members (time), equipment needed...
 - ◆ Meeting the deadlines
 - Product delivery time...

The traditional approach

- ❖ A typical example is a linear programming process.
- ❖ We want to manage the uncertainty as early as possible in the project:
 - ◆ by describing the system functionality in the analysis phase.
 - ◆ defining structure of the system in the system design phase
 - ◆ define the structure of modules (programs, classes) in the detailed design phase.
 - ◆ and when the uncertainty is minimal we start coding.
- ❖ Uncertainty is reduced gradually in each phase and by each product of each phase (SRS, design plan, component design plan...)
- ❖ Reduced uncertainty contributes to easier implementation better solutions and easier maintenance.

Traditional difficulties

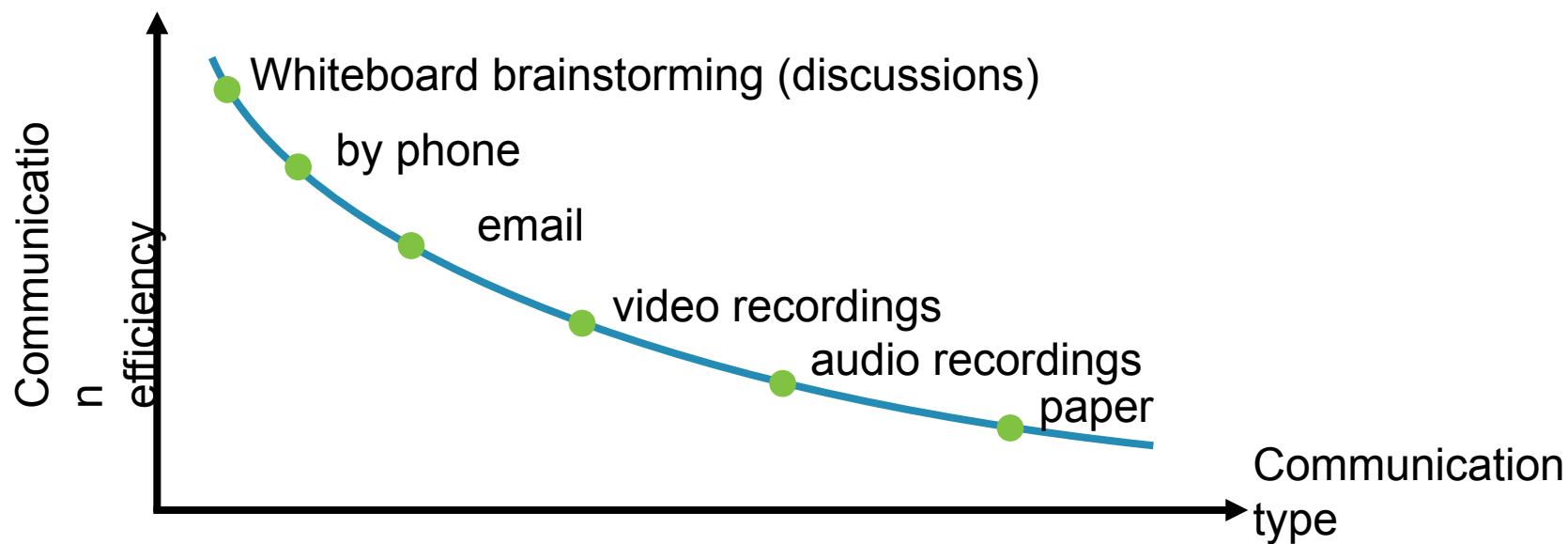
- ❖ The development of virtually every software product faces a series of uncertainties that are difficult to completely eliminate in the traditionally defined phases of a project:
 - ◆ Requirements are only fully clarified in the design or even implementation or testing phase (inconsistencies, deficiencies detected ...)
 - ◆ In each next phase difficulties of the previous one are detected.
- ❖ Consequences:
 - ◆ Products such as SRS, design plans etc. are incomplete and must be supplemented in later phases.
 - ◆ The project scope, timetable, and costs vary as the project progresses!

Alternatives?

- ❖ Why extensive pre-planning and documentation if the right functionality and execution are only proven during coding and testing?
 - ◆ Informal implementation of the analysis and design phases – during coding!
 - ◆ Frequent and automatic testing during implementation!
 - ◆ Involvement of the customer / user in the development process!

Alternatives?

- ❖ Documentation is for communication - why not communicate as efficiently as possible?



- Direct communication, brainstorming, programming in pairs...

Agile approaches

In agile approaches some phases are performed informally in order to

- ◆ increase process flexibility
- ◆ shorten the time to delivery (earlier return of investment, better cooperation with the customer)

Agile approaches replace formality with:

- ◆ informal open relationship within the development team,
- ◆ collaborating within and outside the team,
- ◆ with more active quality control.

The values of agile approaches

- ❖ The individual and the relationship between individuals is more important than the processes and tools.
- ❖ Running software is more important than extensive documentation.
- ❖ Customer involvement is better than contractual negotiations.
- ❖ Responding to change is more important than strictly following a plan.

Principles of agile approaches

- ❖ Divide functionality into small independent components.
- ❖ Frequent releases of new software versions.
- ❖ Changes to requirements are welcome, even if late.
- ❖ Daily cooperation of technical and business staff.
- ❖ Direct involvement of a customer representative in the team.
- ❖ Personal communication (four-eyed) is best
- ❖ Projects should be based on motivated individuals who need to be trusted.
- ❖ Simplicity is a value
- ❖ The teams should get organized by themselves

Some agile methodologies

- ❖ **Extreme Programming (XP)**
- ❖ **Scrum**
- ❖ **Dynamic Software Development Method (DSDM)**
- ❖ **Feature Driven Development (FDD)**
- ❖ **Adaptive Software Development (ASD)**
- ❖ **Crystal Clear**
- ❖ **Rapid Application Development (RAD)**
- ❖ ...

XP - extreme programming

- ❖ *Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation.*

Src: <http://www.extremeprogramming.org/>

<https://www.agilealliance.org/glossary/xp>

https://en.wikipedia.org/wiki/Extreme_programming

XP: extrema

Extrema of extreme programming

- ◆ If code reviews are good, do them all the time (pair programming)
- ◆ If testing is useful, let everyone test all the time (“unit tests”)
- ◆ If simplicity is useful, keep the system as simple as possible, as long as it meets the requirements.
- ◆ If planning is useful, everyone should plan daily (continuous reordering).
- ◆ If architecture is important, it should be planned and improved by everyone.
- ◆ If integration testing is important, the integration and testing should be performed multiple times a day – continuous integration.
- ◆ If small iterations are good, keep them really small (hours rather than weeks)

XP: project execution

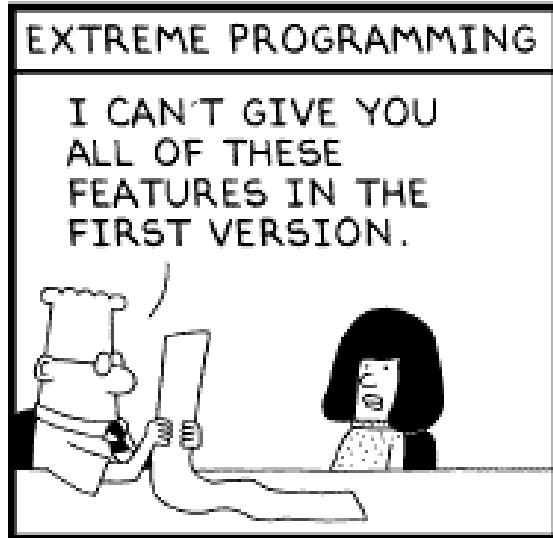
- ❖ Defining stories - the attributes that a customer / user wants.
- ❖ Estimate duration and cost for each story.
- ❖ Choosing stories for the next release / version.
- ❖ Define tasks of each next release.
- ❖ Preparation of test cases for each task.
- ❖ Programming in pairs. No specialization from developers. Presence of customer representative.
- ❖ Continuous integration.

XP: principles (1-6)

1. “Planning game”: Setting the goals of the next version by considering business needs and technical assessments.
2. “Small releases” - A simple system is released and is upgraded with small and frequent updates (new releases / versions).
3. “Metaphor” - a simple story about the functioning of the final system to guide the development.
4. “Simple design” - the system must be designed as simply as possible (unnecessary complexity must be removed as soon as it is identified)
5. “Testing” - testing is happening all the time, “unit tests” are the basis for programming functionality.
6. “Refactoring” - continually rearranging the structure of a system to reduce complexity and avoid code duplication.

XP: principi (7-12)

7. “Pair programming” - always two programmers at one computer.
8. “Collective ownership” - a common source code of the entire system that anyone can change at any time.
9. Continuous integration - changes are integrated into the system several times a day, after each task is completed.
10. 40-hour week - never work more than 40 hours a week.
11. “On-site customer” - The customer representative is included in the development team in terms of continuous availability to answer developer questions.
12. “Coding standards” - strictly following the rules of programming, with an emphasis on how to communicate through source code.



Copyright © 2003 United Feature Syndicate, Inc.

0

XP: additional resources

- ❖ Internet:

- ❖ <http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap>
- ❖ http://en.wikipedia.org/wiki/Extreme_programming
- ❖ <http://www.extremeprogramming.org/>

- ❖ Video:

- ❖ <http://vimeo.com/7849514>
- ❖ https://www.youtube.com/watch?v=LkhLZ7_KZ5w

Scrum



Scrum: roles 1/4

- ❖ Scrum defines the skeleton of a software process in which:
 - ◆ **Scrum Master** ensures the smooth execution of the process (the role of project manager). NOT a team leader!
 - ◆ **Product Owner** represents the interests of the customer and takes care of the proper direction of the project (business, choice of functionality ...).
 - ◆ **Team**, a multidisciplinary team of 7 (+/- 2) members, whose task is to analyze, design, implement, test ...

Scrum: roles 2/4

- ❖ External actors that influence the scrum process:
 - ◆ **Stakeholders** - customers, users - are the ones who will benefit from the project and justify its implementation. Sprint reviews are included in the process
 - ◆ **Management** (of the organization) - which establishes the conditions for project implementation.

Scrum: roles 3/4

- ❖ “pig” roles
 - ◆ scrum master
 - ◆ team members
- ❖ “chicken” roles
 - ◆ product owner
 - ◆ management
 - ◆ stakeholders

Scrum: roles 4/4

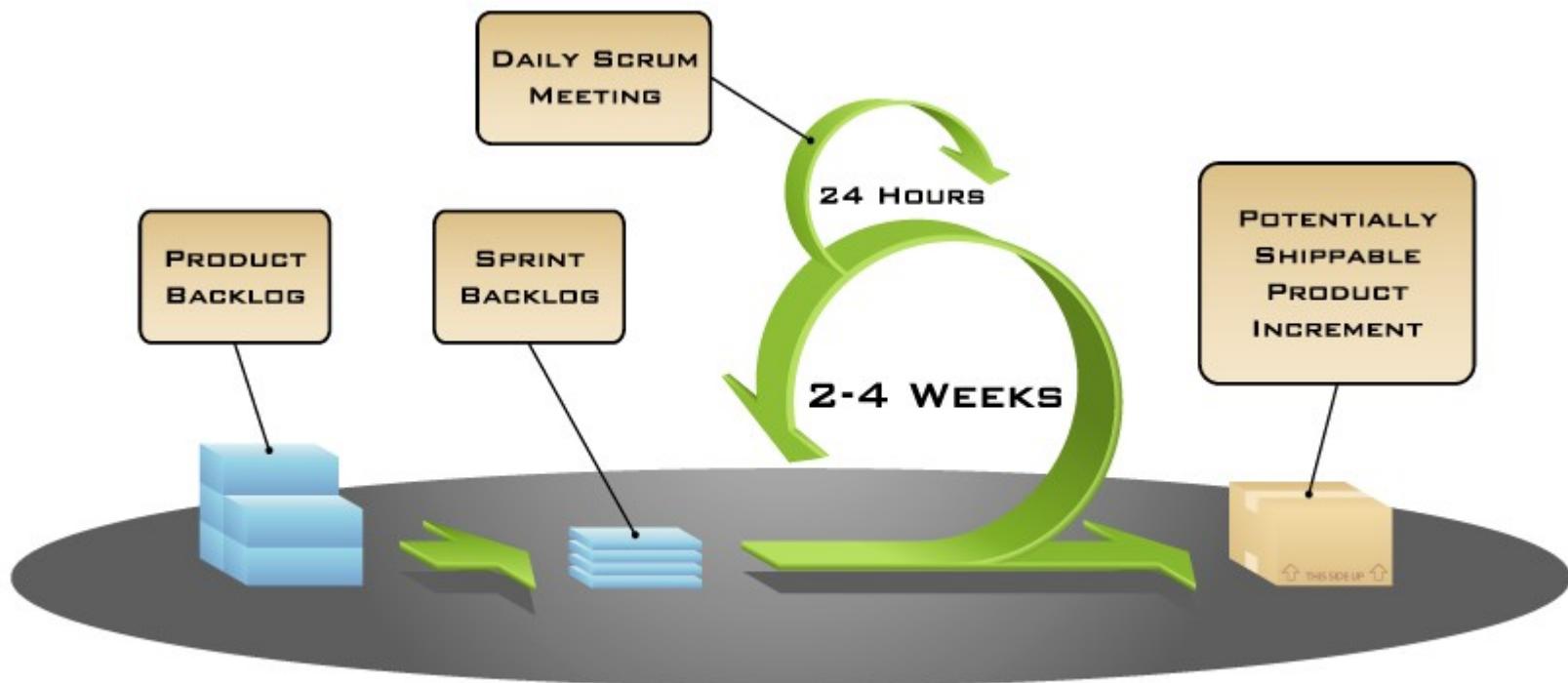
Explanation of roles:

- ❖ *A pig and a chicken are walking down a road. The chicken looks at the pig and says, “Hey, why don’t we open a restaurant?” The pig looks back at the chicken and says, “Good idea, what do you want to call it?” The chicken thinks about it and says, “Why don’t we call it ‘Ham and Eggs’?” “I don’t think so,” says the pig, “I’d be committed, but you’d only be involved.”*

Scrum: the project course

- ❖ The owner of the product arranges and takes care of the product backlog -wish list or features of the software product.
- ❖ Specify the release backlog as a subset of the product backlog. For each feature the execution time needs to be estimated.
- ❖ The release backlog is distributed among several (typically 4-12) parts, so that each part takes up to 4 weeks to complete. These parts are called "sprints" and the related parts of the backlog is called "sprint backlog".

Scrum: illustration



COPYRIGHT © 2005, MOUNTAIN GOAT SOFTWARE

Scrum: sprint

❖ Sprint:

- ◆ In each sprint, the team implements a potentially deliverable product.
- ◆ The sprint backlog must NOT be changed during the sprint run (fixed requirements).
- ◆ At the end of the sprint, the team demonstrates a developed solution.

Scrum: meetings 1/5

- ❖ Declared meetings:
 - ◆ Sprint Planning Meeting
 - ◆ Daily Scrum
 - ◆ Scrum of scrums
 - ◆ Sprint Review Meeting
 - ◆ Sprint Retrospective

Scrum: meetings 2/5

- ❖ Sprint Planning Meeting
 - ◆ At the start of the sprint
 - ◆ Choosing the scope of work (what to do)
 - ◆ Preparing a Sprint Backlog with estimates of the required time (regarding the team).
 - ◆ The meeting is limited to 8 hours.

Scrum: meetings 3/5

❖ Daily Scrum

- ◆ Every day during the sprint period.
- ◆ A quick standing meeting on the status of a sprint.
- ◆ Always starts at the same prescribed time and place.
- ◆ Everyone is welcome, only the pigs have the word.
- ◆ Meeting time limited to 15 minutes.
- ◆ Each team member answers three questions:
 - What did you do until yesterday?
 - What are you planning to do today?
 - Do you have any problems that hinder you from completing the task?

Scrum: meetings 4/5

❖ Scrum of scrums

- ◆ Every day after the daily scrum
- ◆ Meeting between representatives of several teams of the same project.
- ◆ Each team is represented by one member.
- ◆ The content is the same as in the daily scrum:
 - What has your team done since our last meeting?
 - What will your team do until our next meeting?
 - Is there anything that hinders you from completing the task?
 - Do you see the need for teams to work together?

Scrum: meetings 5/5

❖ Sprint Review Meeting

- ◆ Review of completed and unfinished / unfinished work.
- ◆ Presentation of results to stakeholders (demo).
- ◆ Four hours limit.

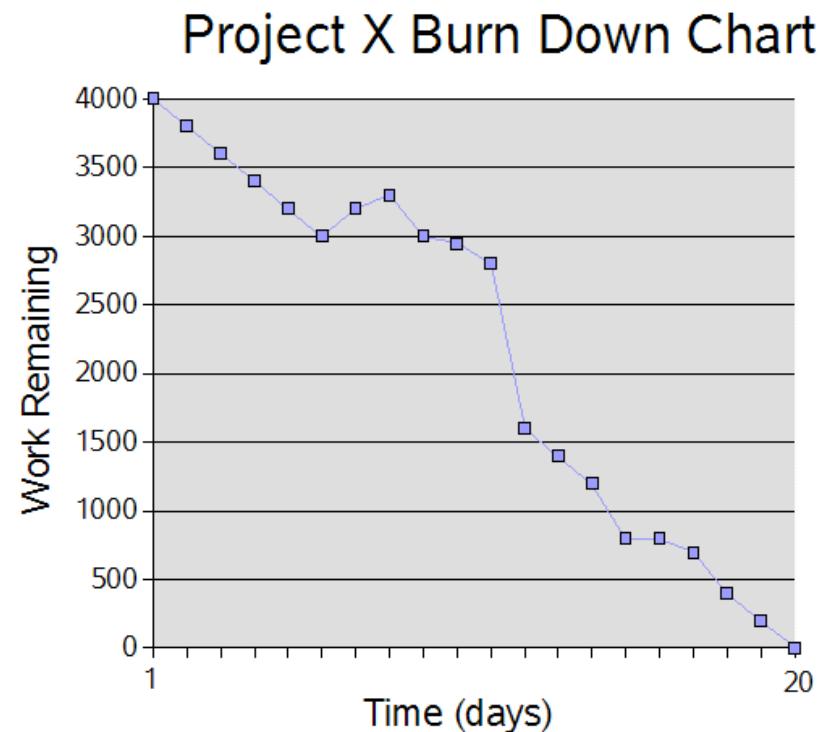
❖ Sprint Retrospective

- ◆ All team members present their views on the completed sprint.
- ◆ Care for continuous improvement of the software process.
- ◆ Answer two questions:
 - What went well in the sprint?
 - What could be improved in the next sprint?
- ◆ Three hours limit.

Scrum: burn down chart

Burn down chart:

- ❖ A publicly released graph showing the remaining work required to finish the sprint backlog.
- ❖ Updated daily.
- ❖ It offers easy insight into sprint progress.



Scrum: resources

- ❖ **Internet:**

- ❖ http://en.wikipedia.org/wiki/Scrum_%28development%29

- ❖ **Video:**

- ❖ <http://www.youtube.com/watch?v=Q5k7a9YEoUI>

- ❖ <https://www.youtube.com/watch?v=XU0IIRItyFM>

General on agile development

- ❖ Agile approaches do not mean ad-hoc programming!
- ❖ They specify the standards to be followed:
 - ◆ coding style,
 - ◆ a way to integrate new functionalities and changes,
 - ◆ testing method (unit testing and integration)
 - ◆ higher level software process (requirements collection and functionality selection, releases etc.)

The success of agile projects

- ❖ It is adversely affected by:
 - ◆ The size (> 20 developers)
 - ◆ Geographic distribution of the development team
 - ◆ Management culture of the company
 - ◆ Morced use of agile approaches
 - ◆ Critical tasks where errors are not permissible.

Advantages of agile methods

- Customer satisfaction by rapid, continuous delivery of useful software.
- People and interactions are emphasized rather than process and tools. Customers, developers and testers constantly interact with each other.
- Working software is delivered frequently (weeks rather than months).
- Face-to-face conversation is the best form of communication.
- Close, daily cooperation between business people and developers.
- Continuous attention to technical excellence and good design.
- Regular adaptation to changing circumstances.
- Even late changes in requirements are welcomed.

Agile approaches' critics

- ❖ Insufficient planning.
- ❖ Insufficient documentation.
- ❖ Urgent above average programmers.
- ❖ Difficult financial planning.
- ❖ More difficult quality assurance.
- ❖ ...

Drawbacks of agile methods

- ❖ Extensive documentation is missing.
- ❖ Extensive testing is useful, but not everything is testable.
- ❖ Difficult to track fast changing requirements.
- ❖ Continuous code refactoring may introduce errors.

Drawbacks of agile methods

- ❖ **Extensive documentation is missing.**
- ❖ The software product must be maintained
 - ◆ Some software products have been around for decades!
 - ◆ Even after the development is complete, we need knowledge of the structure of the system and knowledge of the reasons for the decisions made.
 - ◆ How to maintain products in the event of fluctuation?

Drawbacks of agile methods

- ❖ **Continuous code refactoring may introduce errors**
- ❖ The modifications are meant to improve software performance and as code refinement.
- ❖ The "suboptimalities" of the code may be due to the specific requirements.
- ❖ Changing the software operation (algorithm change) cannot be automated, so it is subject to errors!

Drawbacks of agile methods

- ❖ **Not everything can be tested.**
- ❖ Building automated tests is useful, but tests are always limited.
 - ◆ When is testing sufficient?
 - ◆ Testing sequential programs is difficult, but what about in the case of parallelisms?
 - ◆ How to (automatically) test the quality of security-critical applications?

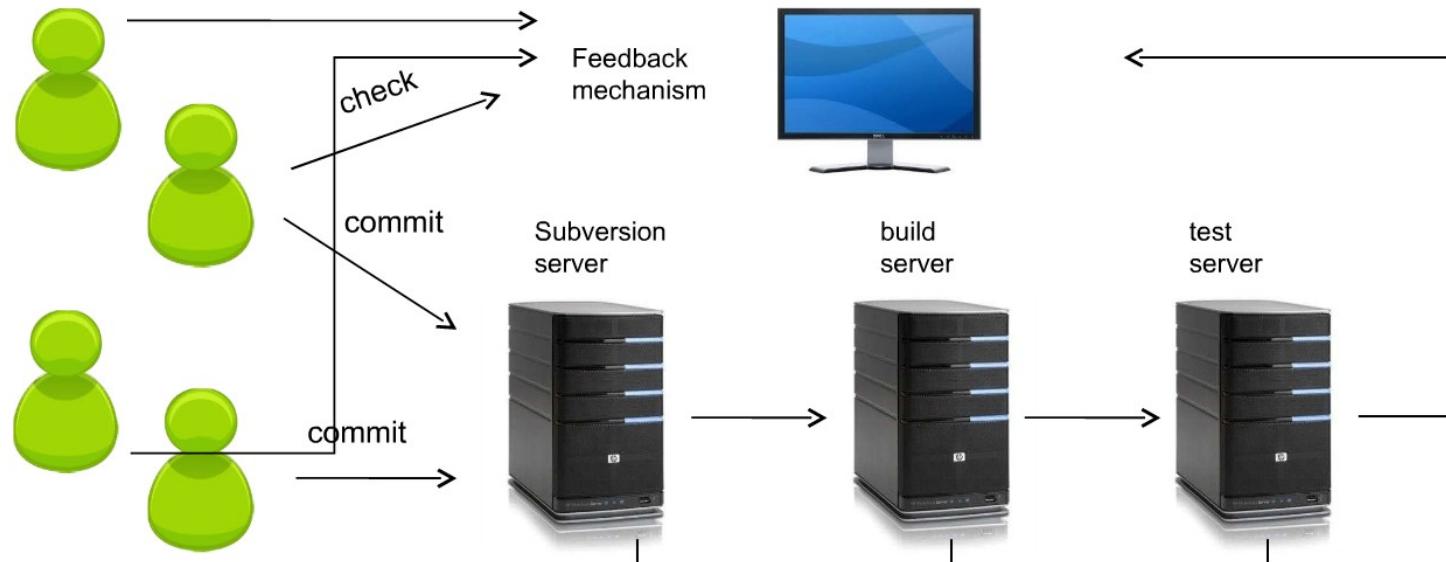
Drawbacks of agile methods

❖ **Difficult to track fast changing requirements**

- ◆ How to manage inconsistent requirements?
- ◆ Why was the change made?
- ◆ Where does the change request come from?
- ◆ Was the change justified?

Continuous (des)integration?

❖ (Continuous Integration – CI)



- ❖ The success of integration depends on:
 - ◆ frequent updating of the source code,
 - ◆ frequent product build,
 - ◆ immediate testing,
 - ◆ immediate debugging.

Agile and formal methods

- ❖ Agile approaches may be more optimal than formal (depending on project characteristics).
- ❖ Agile approaches cannot completely replace formal approaches!
- ❖ Agile and formal approaches can be complementary.

SW processes in general

- ❖ The processes are more or less adaptive or predictive.
 - ◆ Agile processes are more adaptive, but they are more difficult to be predicted for the future.
- ❖ The processes are more or less optimized.
 - ◆ Formal processes are more optimized, but they have more problems coping with changes.

How heavy the methodologies are

