1. Introduction

A term Software Engineering was first introduced in the first NATO conference 1968. The term was defined as follows:

The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

(P. Naur, R. Randell (eds.): Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee, 1968)

Features of Software Engineering

The definition was ahead its time and it is still valid. Software engineering
– is disciplined engineering work,
– offers means to build high-quality efficient software at affordable prices, and
– offers task allocation and tools for all software building phases.

Software and computer-based system

A software is a set of cooperating computer programs, their data files, and documents related to them.

A computer-based system is a set of components that are related to each other.
– hardware and software components,
– users and use environments.

Subsystems and users

Usually a computer-based system (from now on, just a system) consists of several subsystems.
– Each subsystem is an independent entity that cooperates with other subsystems.
– A subsystem consists of hardware and software components that together implement the subsystem functionality.
– The users interact with the system via one or more subsystems.

On building system and software

Building the system and its software is a joint project:
– Software engineering needs knowledge of the environment of the system.
– Cooperation requires software engineers to have skills to communicate with the experts of the application area.
– It is easy to blame software when something goes wrong, even if the actual reason for the failure is something else.

What is required from good software?

Good software has the following essential attributes:
– Maintainability: it is possible to update the product to fulfill changing requirements.
– Reliability: the product does not cause physical or economical damages in any situation.
– Efficiency: the product does not waste resources.
– Usability: the product is logical and easy to use.
2. History of Software Engineering

- The first commercial software was built as early as 1951, in England. The company behind the software was J. Lyons company.
- In the early days software creation was straightforward and uncontrolled. Still, even at 50's software was already relatively versatile.

1960’s and software crisis

- In the early 1960’s a few modern terms, such as testability and interfaces, were defined. Still, at the time software creation was not really Software Engineering.
- During the decade the average size of software projects grew and problems got worse. Projects started to miss their budget and schedule. This was called the software crisis of 60’s.
- Suddenly straightforward coding was no longer enough.

The first Software Engineering conferences

- Finally 1968 NATO organized the first Software Engineering conference where the term “Software Engineering” was introduced.
- The next year NATO organized the second Software Engineering conference. Together in these two conferences the basis of Software Engineering were defined.

Results of the first conferences

- The results of the first conferences may not have solved the software crisis.
  - Some experts think that even today the software crisis remains unsolved.
- Instead the ideas presented in the conferences built a solid foundation for modern Software Engineering.

1970’s and process models

- The first Software Engineering process model, the Waterfall model, was defined in early 1970’s.
  - A process model is a general regulation for engineering software that can be used on different types of projects.
- The basis of the waterfall model was the systems engineering model that was defined for hardware engineering fields.

More process models

- The systems engineering model is sufficient for example for designing bridges. The model was copied directly to Software Engineering.
- Soon software engineers noticed that the waterfall model is not suitable for building all types of software. Thus, in 70’s and 80’s, several new software process models were introduced.
Early days of programming languages
- The first programs were written in a machine language, but the first high-level programming languages were introduced already at 50’s.
- In the early 60’s, the first efficient programming languages Algol, Cobol, and PL/1 were introduced.
- A remarkable step was the C-language from the late 60’s.

1980’s and object-orientation
- The biggest new thing in 1980’s was object-orientation. The first object-oriented conference OOPSLA was held in 1986.
- The principles of object-orientation had of course been introduced earlier, but the first object-oriented languages gathered the principles together.
- Object-orientation spread slowly but steadily from programming languages to all Software Engineering fields.

1990’s and personal computing
- In the early 90’s, Software Engineering was quite stable. A new trend personal computing (everyone had a private computer) changed the needs for software but not the techniques of building them.
- This was the era of tools. CASE-tools (Computer-Aided Software Engineering) evolved fast.

1999 and agile process models
- The latest big change in Software Engineering occurred 1999 when the first agile process model Extreme Programming (XP) was introduced.
- Agile process models soon became popular among programmers and small and medium companies because the models preferred programming to design.

Today (2007)
- Currently (2007) Software Engineering is best described with terms distribution and specialization.
- Although the basic principles of Software Engineering (those that we teach in this course) have not changed much, companies use their own tailored process models, methods and tools.
- The customers expect results as early as 3-4 months after the project kickoff.
- A very important aspect is good knowledge of the used Application Interfaces (APIs).

Specialized today (2007)
- Also built software differ more than ever.
  - First, we have small software for cellular phones and mobile equipment.
  - Second, we have huge software with very clearly defined high performance and safety requirements.
- Software companies need to specialize in order to keep their market shares.
Future

- Software Engineering will face three challenges in the future:
  - Heterogeneity challenge.
  - Delivery challenge.
  - Trust challenge.
- The challenges are not independent. Some may even conflict with each other.

Heterogeneity challenge

- Software-based systems have to work in various environments cooperating with different types of systems.
- Software has to be built to work both with current and with legacy systems.
- Legacy systems often need new functionality.
- The life span of old software is often expanded beyond its natural limit.

Delivery challenge

- Software development needs a lot of time. Time is an expensive resource.
- In business one has to be dynamic and ready for changes.
- Thus, also software has to be built fast and be ready for change without compromising from quality.

Trust challenge

- Software systems affect more and more our lives. The more they control, the more we need to trust them.
- Trust does not come easily. One has to be able to figure out from software that it works as planned.
- For example, modern cars have surprisingly lot small software-based problems.

3. Software process

- A software process describes a set of activities that lead to a developed software.
- A process is a set of regulations. It explains activities and relationships.
- An instantiation of a process is a project. It implements the activities of a process.
- In practice, each software company uses processes tailored for their specific use.

Process model

- A process model is an abstract description of a process.
  - A process model may be a source for various processes in different organizations and application domains.
- The process model describes general principles, not details. Thus, the process model defines frames inside where processes will be built.
Basic activities of process models

- Each process model defines the following basic process activities:
  - Software specification (i.e. requirements engineering).
  - Software design.
  - Software implementation and unit testing.
  - Software integration and system testing.
  - Usage and maintenance.

Requirements engineering

- What requirements does the software have?
  - What functionality must it include?
  - What quality properties must it fulfil?
    - such as speed and usability requirements.
  - What restrictions does the software have?
    - such as cooperation with other software.

Requirements engineering process

Software design

- A software design is a description of the structure of the software to be implemented, the data which is part of the system, the interfaces between system components, and the algorithms used.
- Software design is based on the gathered requirements and results one or more software models for the implementation phase.

Design levels

- The design process requires several phases that process the problem at different abstraction levels. The levels may for instance be as follows (or something else; it is not wise to unify the design process too much):
  - Architectural design
    - Define a high-abstraction model of the system showing subsystems and used components.
  - Abstract specification
    - Define services and restrictions for each subsystem.
  - Interface design
    - Define interfaces between subsystems.
  - Component design
    - For each subsystem define components that implement the subsystem’s offered services and interfaces.
Design levels 3

- Data structure design
  - Define data structures that are necessary for the implementation.
- Algorithm design
  - Define algorithms that are necessary for the implementation.
  - A component is a distinct element of a software that cooperates with other components. Together components implement the subsystem functionality.

General design process

Implementation and unit testing

- In implementation, a set of components is created from the design.
  - A software component has a clear interface to outside. It does one thing very well (and nothing else).
  - A component may vary from a small object to a complete computer program.
- Unit testing, as part of implementation, is to ensure that components behave as promised.

Integration and system testing

- Software components are integrated together.
- Component cooperation is tested in integration testing.
- Components are integrated to independent subsystems.
-Subsystem cooperation is tested.
- The complete system is tested in a real environment: system testing.

Testing phases in the software process

- In the figure, testing is below and test design is above.
- The figure is a variation of the V-model of testing. It will be covered later.

Software usage and maintenance

- It is often necessary to change software after it has been delivered to the customer.
  - Software usage reveals errors.
  - User requirements change or get more detailed.
  - System environment and computer equipment change.
  - System usage patterns change.
- Maintenance = changing software after it has been delivered.
- Evolution = Inevitable elements that force software to change.
3.1. Most common process models

- The most common process models are:
  - Waterfall model: linear stages.
  - Evolutionary development: prototypes.
  - Reuse-based process models: reuse earlier implemented software components.
  - Incremental delivery: build small pieces.
  - Spiral development: repeat same tasks on different scale on different stages.
  - Agile process models: less management.

Waterfall model

- The waterfall model has a linear approach:
  - A finished stage is never re-entered.
  - Each stage is finished, results accepted and frozen before moving to the next stage.
- The stages of the model are the basic activities of process models.

Waterfall model stages

Waterfall model advantages and disadvantages

+ The model is easy to understand and learn.
+ A cost estimate and a schedule are easier to create than in other models.
+ The developed software architecture is usually solid.
+ The resulting good documents help in the maintenance stage.
- It is difficult to find out all needed requirements at once and the requirements may change.
- A clear linear process is usually not an option. Some iterative issues always exist.
- The development stage lacks a customer representative.
- The life cycle of errors can be very long.

Evolutionary development

- In the evolutionary development model, the software is created via intermediate versions:
  - Intermediate versions are prototypes that are developed step by step to the final deliverable software.
  - The first prototypes are fast “easily glued” quick implementations.
  - Customers give feedback of the intermediate versions.
  - Received feedback defines the basis of the next version development.
What to do with the prototypes?

- **Discard prototype**
  - Prototype is only for refining wishes and requirements.
- **Further develop prototype**
  - Prototype is for showing core functionality of the software.
- **Why a prototype is not the final software?**
  - The prototype is a partial implementation.
  - The prototype does not fulfill quality requirements.

Evolutionary development advantages and disadvantages

- Continuous feedback from the customers.
- Usually a short error life span.
- The user interface is present from the early development so the learning curve of the software should be gentle.
- Continuous changes break the software architecture.
- A prototype development software is a must.
- It may be difficult to ensure the customer that a middle prototype is not sufficient for a final product.
- The last prototype is in fact the final product so the project does not have a clear finish time.

Reuse-oriented process models

- The goal is to use as much reusable software components as possible.
- Reusable components may be found
  - from earlier projects,
  - from a store (so called cots-components (commercial-of-the-shelf components),
  - from Free Software projects (dangerous since there is no maintenance guarantee).
- Reusable components are stored in a component library.

Reuse-oriented process model stages

- The stages of processes are as follows:
  - Requirements elicitation and analysis
    - Collect and analyse requirements.
  - Component analysis
    - Seek suitable components from the component library for the found requirements.
  - Requirements modification
    - Modify requirements to reflect the available components.

Reuse-oriented process model stages II

- Stages continue:
  - System design with reuse
    - The designers take into account the reused components and seek new reusable design component candidates.
  - Development and integration
    - Components and designed code is integrated to create the new system. New reusable implementation components are seek.
  - System validation
    - Ensure that the product fulfils customers’ needs

Reuse-oriented process advantages and disadvantages

- Component reuse can save a lot of work
- Component-based design leads to reusable solutions.
- We can make profit with the contents of our component library.
- We don’t need to reinvent the wheel in every project.
- It is possible to use too much reuse techniques and thus try to adapt customer requirements to components by force.
- The component documentation may be incomplete.
- Adding a component to a software is sometimes surprisingly difficult.
- It is often very difficult to correct a bug from a finished (commercial) component.
Incremental delivery

• In the waterfall model, a software is specified and designed as a monolithic entity. Quite often this does not work:
  – requirements are not well known,
  – requirements are not well defined, or
  – requirements change during the project.
• In incremental delivery, the software is specified and implemented iteratively in cycles: from core to special functions.

Incremental delivery II

• The developed software is divided into fragments that are implemented separately in cycles.
  – The software core is implemented first
  – New functionality is added to the core in the following cycles.
• On each cycle, only requirements relevant to the cycle are processed.
• Requirements from yet unimplemented cycles may change.

Incremental delivery strategy

Spiral development

• In the spiral model, each spiral represents a stage in the process.
• Each spiral has four sectors:
  – Objective setting and schedule
  – Risk assessment and reduction
  – Development and validation
  – Result feedback and next spiral planning
• Spirals need not to follow the same process model: the spiral model is a kind of a meta-model for process models.

Spiral development advantages and disadvantages

• The model is suitable for tasks of different sizes.
• Each spiral may have its private process.
• Risk management is part of the process model.
• The model is flexible enough to model the complete life span of a product including maintenance.
  – The model may be difficult to explain to a non-expert.
  – Project management may be difficult.
  – Schedule planning becomes more difficult.
  – Practical experiences of the spiral development model are mostly lacking: agile process models have stolen interest from the model.

Agile process models

• The process models of 1980’s and 1990’s were based on
  – careful project planning,
  – formal quality assurance,
  – detailed analysis and design methods,
  – CASE tools, and
  – rigorous closely controlled software processes.
Dawn of agile process models

- The process models supported especially development of large long life-span software, but they turned out to be too inflexible for small and medium-size software projects.
- Due to this issue a set of agile process models were born. The models emphasized on the software itself instead of detailed design and documentation.
  - Currently the most used agile process models are eXtreme Programming (XP) and Scrum.

The nature of agile process models

- All agile process models are iterative.
- The length of an iteration cycle is only a few weeks. All stages from requirements engineering to delivery are executed in that time.
- Due to a short cycle and a strong need for feedback, a customer has to be present in the development process.

Agile process model principles

- Active customers
  - Customers are actively involved in development. The customers introduce and prioritise new system requirements and evaluate results of the latest cycle.
- Incremental functionality
  - New functionality is implemented to the software on each cycle. The functionality is based on the customers’ feedback.

Agile process model principles II

- People are more important than process
  - Abilities and working styles of project members are more important than a heavily controlled process.
- Accept change
  - Requirements change so the system may need to be refactored.
- Keep it simple
  - Both the software and the process must be as simple as possible (but not any simpler). Documentation is minimised.

Agile process model advantages and disadvantages

+ Requirements analysis, design and unit testing are easy to execute in a short cycle.
+ A customer is always present to give feedback.
+ Agile process models are flexible and favour people more than management.
+ Requirements are allowed to change during the project.
+ Small and medium-size software development is efficient.
  - It is very difficult to create a schedule for the complete project.
  - Project management is difficult.
  - Continuous code refactoring is expensive and may weaken product quality.
  - Sometimes even the smallest possible indivisible entity is not small enough for a single cycle.
  - It may be tempting to fill a cycle with more work than what actually is possible to finish in time.
  - Large software development is difficult.

Extreme programming (XP)

- Software is developed in very small cycles.
  - A small core is built first
  - On each cycle, the smallest possible sensible set of requirements is implemented on top of the core.
  - The implementation order depends on customers’ needs.
  - The project is planned as short a period ahead as possible.
  - Documentation is pruned to minimum. A working product is more important than a heavy document library. Program code is the most important document.
  - The model includes a set of process rules (principles).
4. Requirements engineering

• Perhaps the most demanding issue of software engineering is to find out the customers’ requirements for the system.
  – All easily defined systems have already been specified so the issue is likely to get worse in the future.
• The requirements describe what kind of a system the customers want to aid or simplify their tasks or actions.

Requirements engineering tasks

• Requirements engineering is a process for finding out the requirements of the developed system.
• In requirements engineering, we want to find out
  – what the system has to offer (services),
  – what attributes affect the complete system and how (quality attributes), and
  – what limiting factors the system has.

4.1. Requirements engineering process

• Like in general processes, also requirements engineering processes have a lot of variation. In practice, all processes consist of the following basic process activities:
  – feasibility study,
  – elicitation and analysis,
  – specification, and
  – validation.

Requirements engineering activities (same as slide 27)

• Requirements engineering may also benefit from a spiral-like view.
• A spiral shows a clear implication that not all requirements may be elicited at once but instead several iterations are necessary.
• The requirements spiral also allows other activities such as design/implementation to be parallel with requirements engineering.
Feasibility study

- Feasibility study is a short focused study before requirements engineering process that aims to answer a number of questions:
  1. Does the system contribute to the overall objectives of the organisation?
  2. Can the system be implemented using current technology and within given cost and schedule constraints?
  3. Can the system be integrated with other systems which are already in place?

Moreover, although Sommerville does not mention it:
- Is it reasonable to implement the system, or is it possible to buy an already developed system from somewhere? Is the already developed system suitable for customer's needs?
- The result of the feasibility study stage is the decision of whether it is worth to develop the system.

Requirements elicitation and analysis

- In the requirements elicitation and analysis stage software engineers work with customers and system end-users to find out
  - what services are required from the system,
  - what sort of performance is required from the system,
  - what hardware and domain constraints must be noted etc.
- Stakeholders are involved in the process.

Stakeholders

- A Stakeholder is a person or group who will be affected by the system, directly or indirectly.
- Different stakeholders have different needs for the developed system É conflicts.
- Stakeholders are sometimes unable to express of what they want from the developed system.

Elicitation and analysis stages

- The following stages are present in the elicitation and analysis:
  - Requirements discovery.
  - Requirements classification and organization.
  - Requirements prioritisation.
  - Requirements negotiation.
  - Requirements documentation.

Requirements elicitation and analysis process

- Requirements elicitation and analysis is an iterative process.
- The process is started from large requirements and on each spiral smaller and smaller details are analysed.
Requirements discovery

- Requirements discovery is the process of interacting with stakeholders to collect their requirements.
- The discovery process may be direct (interviews etc.) or indirect (reading documentation, ethnography etc.).

Requirements discovery techniques

- The following techniques have turned out to be useful in requirements discovery:
  - Viewpoints: find out what kind of direct or indirect relationships stakeholders have with the system.
  - Interviews: stakeholders are asked specific questions of what they expect from the system.

Requirements discovery techniques 2

- More techniques:
  - Scenarios and use cases: a scenario is a real-life example of how to use the system. In a complex system, related scenarios are grouped together to use cases.
  - Ethnography: immerse in the working environment where the system will be used and observe the day-to-day work and the actual tasks of system end users.

Requirements specification

- In requirements specification, found requirements are classified to user and system requirements.
- The classified requirements are modified and expanded, when necessary. User requirements can be used to define new system requirements.
- The result of the stage is a list of prioritised user and system requirements for the entire system.

User requirements

- User requirements are natural language statements and suitable diagrams of
  - what services the system is expected to provide, and
  - the constraints under which it must operate.
- The readers of user requirements are not software professionals so the statements must be easy to understand.

System requirements

- System requirements give detailed descriptions to requirements in a joint structural description technique:
  - system's functions,
  - system's services, and
  - system's operational constraints.
- The system requirements give exact definitions of what the system does and what constraints it has.
User and system requirement usage

• User requirements are for
  – deciding who will get the software development contract and at what cost,
  – describing the system to end users,
  – describing the system to our high management staff.
• System requirements are for
  – defining a contract between the software development company and the customer,
  – input to the design stage of the software.

Requirements classification

• User and system requirements are classified as follows:
  – Functional requirements are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
  – Sometimes functional requirements may also explicitly state what the system should not do.

Requirements classification 2

– Non-functional requirements are constraints on the services or functions offered by the system.
– Non-functional requirements affect the whole system. The functional requirements of the system must follow the non-functional requirements.
– Sometimes non-functional requirements affect the software development process and its techniques.

Requirements classification 3

– Domain requirements come from the application domain of the system and that reflect characteristics and constraints of that domain
– Domain requirements may define relationships to other systems or constraints from laws of physics.
– Domain requirements may either be functional or non-functional.

Functional requirements

• Functional requirements may be modeled as
  – services: what does the system offer,
  – functions: what does the system do and how,
  – both: what commands implement what services (cross-checking).
• Functional requirements must be
  – complete: all services are defined,
  – consistent: no conflicting definitions.

Non-functional requirements

• Non-functional requirements are at least as important as functional requirements. They define how the user feels the system.
• It is possible that the complete system is useless if a non-functional requirement is not met.
• Organizational non-functional requirements affect e.g. maintenance.
Types of non-functional requirements

- Performance requirements
- Space requirements
- Usability requirements
- Efficiency requirements
- Reliability requirements
- Portability requirements
- Interoperability requirements
- Ethical requirements
- Legislative requirements
- Implementation requirements
- Standards requirements
- Delivery requirements
- Safety requirements
- Privacy requirements
- Product requirements
- Organisational requirements
- External requirements

Domain requirements

- Domain requirements are important since they define constraints that the system must follow in order to cooperate with other systems or under laws of physics.
- Examples of domain requirements are interfaces to other systems and environmental constraints.

Requirements validation

- The goal of requirements validation is to compromise conflicting requirements and to show that the specified system is what the customer wants.
- Validation is very important because faulty requirements affect the system during its full life cycle.
  - The later a faulty requirement is found, the more expensive it is to repair.

Requirements reviews

- Requirements reviews check for
  - Consistency: are requirements described in a coherent manner and with proper techniques?
  - Completeness: do requirements describe the complete system?
  - Verifiability: are requirements testable?
  - Comprehensibility: do the procurers or end-users of the system properly understand the requirements?
  - Traceability: are the origins of the requirements clearly stated?
  - Adaptability: can the requirements be changed without large-scale effects on other requirements?

Software requirements document

- Software requirements document (or software requirements specification, SRS) gives an unambiguous description of
  - the purpose of the system,
  - the vocabulary used in the documentation,
  - user and system requirements,
  - system architecture and system models
  - system life cycle and its relationships to other systems.

Document structure

- The structure of the requirements document must be suitable at least for
  - being a contract between the customer and the development company,
  - the basis of design,
  - an introduction of the software to the high management staff, and
  - the basis of system testing.
5. Software design

• In the design stage the logical structure of the software is decided.
• The structure is described in various abstraction levels.
  – The higher is the abstraction level, the better general view it offers but details are hidden.
  – The highest abstraction level of design is an architectural plan. The lowest level is pseudo code.

Design activity

• Software design is a creative activity that should not be especially controlled.
  – An opposite view, heavily controlled design process, has gained endorsement.
• There is no easy straightforward way to design a software. Each design process is unique.
• A good basis for a software design is for example the process model on slide 32.

5.1. Architecture design

• Large systems are always divided into cooperating subsystems.
• Architecture design is a stage where subsystems are identified and communication between subsystems is defined.
• The output of the architecture design stage is an architecture plan.

Subsystems and non-functional requirements

• The system’s division to subsystems depends on non-functional requirements:
  – If throughput is important, the best solution is to use a small number of large subsystems.
  – If security is important, a layered architecture is a good solution. The most security-critical functions are at the deepest layer.
  – If reachability is important, it is worth to replicate services to several subsystems.
  – If maintainability is important, it is worth to use a large number of small subsystems.

Architecture plan

• The deliverable of the architecture design stage is an architecture plan. It includes
  – a description of the complete system,
  – created graphical system models and their textual descriptions,
  – the system’s division to subsystems and of each subsystem its division to components,
  – subsystem-to-subsystem communication methods.

Subsystem division

• The next stage after architecture design is to divide each subsystem into modules.
  – A module is a component of the system that uses/offers services from/to other components.
  – Unlike a subsystem, a module is not an independent entity. It cannot exist alone, but instead it cooperates with other modules to implement subsystem functions
5.2. Interface design

- Interface design is perhaps the most important stage of design. In the stage the interface of each subsystem and module is designed and documented in detail.
- Interfaces must be defined so that any user requiring an offered service need not to know anything about how the service is implemented.

Interface definition

- Interface definitions must be unambiguous and non-conflicting.
- An interface definition must include
  - the services of the interface,
  - the types and value ranges of each input and output of each service, and
  - of each service, the contracts that must be met before using the service and the contracts that the service guarantees to be true after the service call.
- A correct interface abstraction level is important.

5.3. Object-oriented design

- The design process in slide 32 is a generic one. It can be specified for example to an object-oriented approach.
- Sommerville lists five stages that are especially suitable for object-oriented design:
  - understanding, architecture, objects, models, interfaces.

Object-oriented design stages

1. Understand and define the context and the modes of use of the system.
   - That is, understand the requirements of the system.
2. Design the system architecture.
   - Define the big picture: architecture plan.
3. Identify the principal objects in the system.
   - Objects are of different complexity from subsystems to details of a specific service or function.
4. Develop design models.
   -Subsystem and component design.
5. Specify object interfaces.
   - Interface design.

Object design model versus general design model

- Compared to the general design model, the object design model emphasizes on the objects.
- Objects are handy for instance in modelling real world entities, but it is still wise to have the component level in design to collect cooperating objects into larger entities.
- It is possible to identify objects from system models and descriptions. Using the identified object and adding new ones offers means to specify the architecture and system models.

6. Verification and validation

- Verification and validation (V&V) is a stage where it is ensured that
  - the software fulfils its requirements, and
  - the software fulfils the customer's needs and expectations for the software.
- V&V is an active process during the complete life cycle of the software project.
Verification and validation

- In verification, it is ensured that the product fulfils its specification – that is, the documented requirements:
  - “Verification: Are we building the product right?”
- In validation, it is ensured that the product fulfils the needs and expectations the customer has:
  - “Validation: Are building the right product?”

Verification and validation techniques

- In V&V, two techniques are superior to others: software inspections and testing.
  - Software inspections.
    - In a software inspection, a group of people systematically analyse and check in a formal inspection meeting a project output seeking defects and errors. Inspections are a static technique. They do not require an executable program.
    - The inspected product output may for instance be part of the requirements document, design plan, or program code.

Verification and validation techniques 2

- Software testing.
  - In testing, a software or some subset of it is executed with known data. By analysing the results and program execution it is ensured that the behaviour of the software is as expected. Testing is a dynamic technique since it requires an executable program.
  - Software inspections may be used at any stage of the process. Testing is possible only after having a piece of executable code.
  - Both techniques are needed in V&V since they reveal different types of errors.

Goal of verification and validation

- The goal of V&V is to ensure that the software fulfils its expectations.
- The software does not have to be flawless, and usually it is not.
- Instead, the software has to be "good enough" for its defined use.
- How much is "good enough" depends on the system domain.

V-model of V&V (lots of V’s here!)

- The V-model describes the relationship of V&V to the other process stages.
- Inspection meetings may be met at any stage of the V-model or between them.

6.1. Inspections

- An inspection is a formal meeting where a piece of document is inspected. The goal of the inspection is to find defects and errors from the inspected document.
  - Defect = incomplete definition or missing functionality.
  - Error = Wrong definition or unwanted action.
Reviews

- A review is a technique where one or more people seeks deficiencies and errors from someone else’s output.
  - An inspection is the most formal review technique.
  - Other review techniques are for example pair programming, team/technical review, and code walkthrough.
- Reviews can be used at any stage of the process. Any document may be verified or validated with a review.
- Reviews improve product quality. The earlier a deficiency or error is found, the easier it is to correct.

The nature of inspections

- An inspection is a formal meeting where 3-6 persons gather together.
  - The meeting has an exact schedule.
  - The participants of the meeting present different stakeholders from the customer to project members.
  - All found deficiencies and errors are collected.
  - Each participant is required to spend about two hours to review the inspected document in advance.

Participants in the inspection

- Roles of participants:
  - Moderator: is responsible for the schedule and the agenda of the inspection.
  - Scribe: writes down found issues.
  - Reader: introduces the inspected subject.
  - Author/owner: represents the authors and clarify issues when necessary.
  - Inspector: finds deficiencies and errors from the document (everyone’s role).

Before the inspection

- Before the inspection meeting:
  - all documents needed in the inspection are available,
  - participants have had time to read the documents (preparation time, about 2h),
  - participants have received check lists of the most common defects and errors, and
  - the inspected document has no visible defects or errors.

Inspection meeting

- The inspection meeting may at most two hours. During the meeting, the participants concentrate solely on finding defects and errors.
- The participants do not discuss about the defects that have been found. When a defect has been perceived, the secretary records it up and the meeting moves along.

End of inspection

- At the end of the inspection meeting, the team votes for the acceptance of the inspected document:
  - Accepted as is: no changes.
  - Accepted with changes: found defects and errors must be corrected but a new inspection meeting for the document is not necessary.
  - Rejected: found defects and errors must be corrected. After corrections, a new inspection meeting for the document is called.
Golden rules of inspections

1. Evaluate product, not author.
2. Define schedule and follow it too.
3. No argumentation.
4. No solving of found problems.
5. Limit participants to 3-6 people.
7. Use checklists both in preparation and in the meeting.
8. Reserve enough time and resources.
9. Train participants.
10. No open phones in the meeting.

6.2. Testing

• Testing has two goals:
  – To show to the customer and the project that developed software fulfills its requirements. This is validation testing.
  – To find defects from the software that cause the software to malfunction, function incorrectly or not to fulfills its specification. This is defect testing.

Features of testing

– Test design is proper to do in parallel with development using the V-model.
– Test preparation is test environment programming and data gathering; that is, normal software development.
– Test execution must be automated so that it is simple to rerun the tests.
– An oracle is needed in test results analysis. It tells whether the result of the test was right or wrong.

Complete testing is not possible

• Complete testing, where software is tested with all possible inputs, input combinations, and timings, is not possible in practice.
  – Even with a simple software it could take years to execute complete testing.
• Due to this a subset of all possible test cases is selected for testing.

Testing stages

• Sommerville divides testing into two:
  – In component testing, software is tested in small entities. Tested entities are integrated and tested together.
    • Component testing is mainly a development team activity.
  – In system testing, the complete system is tested as a single entity.
    • System testing is mainly an external test team activity

System testing

• The stage is connecting:
  – Components are integrated to each subsystem and the cooperation of components is tested. This is integration testing.
  – The components are added and are tested one at a time until all the components of the subsystem have been connected and have been tested.
  – When all the subsystems have been collected and tested, it is tested that the subsystems act together correctly.
Integration testing

- In integration testing, the cooperation of the ready and separately tested functional components of the subsystem is tested.
- The integration can be top-down (first test managing units) or bottom-up (first test performing units).
- It is also possible to integrate from both directions at the same time.

Stress testing

- After integration testing, it is possible to test critical issues of the system:
  - performance,
  - reliability and robustness,
  - security.
- In stress testing, the tested software is put to its limits and preferably even beyond them.

Acceptance testing

- Acceptance testing (or Sommerville: Release testing) is executed after integration testing is over.
- The purpose of acceptance testing is to ensure that the system is in a condition where it can be delivered to the customer.

Unit testing

- Component testing or unit testing is executed before system testing.
  - Objects that implement the functionality of a component are tested separately and then integrated into the component. After that, the component is tested again as an indivisible entity.
  - Unit testing and programming are related: unit tests are written at the same time the unit under test is programmed.

Component interface testing

- An essential part of the component testing is interface testing. This stage finds defects caused by interface errors or false expectations of interface behaviour.
- The interfaces and interface testing are extremely important because the services of both objects and components are defined through their interfaces.

Interfaces

- Interfaces:
  - Parameter exchange interfaces.
  - Shared resource interfaces: memory etc.
  - Procedural interfaces: service calls.
  - Message interfaces: subsystems.
- Identified faults:
  - An interface is used incorrectly.
  - The interface behaviour is misunderstood.
  - The timing of an interface is wrong.
7. Software Evolution

- When a developed software has been delivered to the customer, its development process is completed. Yet, the product itself is in the early stages of its life cycle.
- During its life cycle the product receives pressures of change:
  - The requirements will change.
  - The business goals will change.
  - New bugs will be found.
  - System hardware will change.
  - System environment will change.
- The pressures will force the product to change: software evolution will be present.

Relieve pressures of change

- The pressures of change are relieved with two approaches:
  - If it is possible to continue the development process, in the next release development process the evolution requirements are implemented.
  - If it is not possible to continue the development process, a stage called software maintenance is needed. There the old version is updated to fulfill new requirements.
- In both cases the result of the process is a new software release where the pressures of change have been notified.

Spiral model of development and evolution

Evolution expenses

- Answering software evolution is expensive. Up to 90% of software development expenses are after the first release.
- The more dynamic is the environment for the software, the more expensive will the evolution become.
  - A software that implements a clearly specified algorithm does not need much maintenance (for instance, a control software for a washing machine).
  - A software that implements a need in a continually changing environment will require continuous maintenance and a rapid release rate (for instance, internet-software).

7.1. Evolution dynamics

- Software evolution is management of change. Lehman and Belady developed a set of laws of the stage. The laws are called Lehman’s laws.
- Lehman’s laws describe software pressures of change and their effects on the software and its stakeholders.
- The laws have not been validated so in fact they are theorems (assumptions).

Lehman’s First Law

- Law of continuing change:
  - A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.
  - The law states that software evolution is inevitable. Software will always need changes during their life span.
Lehman’s second law

- **Law of increasing complexity:**
  - As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.
  - This is a kind of a law of entropy in software maintenance. It states that the structure of software will collapse in time unless extra resources (energy) are reserved for preserving it.

Lehman’s third law

- **Law of large program evolution:**
  - Program evolution is a self-regulating process. System attributes such as size, time between releases and the number of reported errors is approximately invariant for each system release.
  - This is a law of large program maintenance schedules. According to it the maintenance process is defined already in the development stage.

Lehman’s fourth law

- **Law of Organisational stability:**
  - Over a program’s lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.
  - According to Lehman most large software projects will reach an optimal level after which adding new resources or staff has little impact on evolution.

Lehman’s fifth law

- **Law of conservation of familiarity:**
  - Over the lifetime of a system, the incremental change in each release is approximately constant.
  - It is not possible to just concentrate on adding new functionality to a maintained product. After a functional release a new release is necessary to fix errors from the previous release.

Other Lehman’s laws

- The other Lehman’s laws state how customers see software evolution. They can be summarised as follows:
  - The less software evolution is noted, the more unsatisfied its customers will become in time.
  - Lehman’s laws describe well how does software evolution of large tailored software work.

7.2. Software maintenance

- Software maintenance is a general term to a process where a delivered software is modified during its life cycle.
  - Usually a maintenance team working in the customer company is responsible for the maintenance process.
  - Usually maintenance is about making small and medium changes to the software. The architecture of the software does not usually change much.
Types of software maintenance

- There are three types of maintenance:
  1. Maintenance to repair software faults (corrective maintenance).
  2. Maintenance to adapt the software to a different operating environment (adaptive maintenance).
  3. Maintenance to add to or modify the system’s functionality (perfective maintenance).
- Usually a software change requires all three types of maintenance.

Maintenance type shares

- According to several research, the shares of maintenance types are about as follows:
  - Correcting system faults: 17%
  - Changing the system to adapt it to a new operating environment: 18%
  - Implementing new requirements: 65%

Evolution process

- The process of software evolution varies considerably. It depends on
  - the type of the maintained software,
  - the process used in the development phase of maintained software,
  - the level of professional skill of persons involved with maintenance.
- The process may vary from a completely informal to a strictly controlled one.

Evolution process tasks

- Tasks related to evolution processes:
  - Impact analysis (Sommerville: also change analysis)
    - Find out how much a change affects the structure of the changed software and how much its implementation would cost.
  - Release planning
    - Decide what changes will be implemented to the next version. The changes will trigger one of the three maintenance types.

Evolution process tasks 2

- Tasks continue:
  - change implementation
    - Implement the decided changes.
  - system release
    - Deliver a new version of the maintained software.

Software evolution and development process differences

- An evolution process differs from a development process:
  - The evolution process does not start from scratch.
    - The first task is to understand the structure and solutions of the update software.
    - None of the changes may break the existing solutions of the software.
  - An alternative to a software update is a software re-engineering.
Software re-engineering

- If it appears that the maintenance costs of software start to grow very high and still there is no guarantee for a quality result, it may be time to re-engineer it.
- In software re-engineering, a part of the software is re-implemented and documented so that its maintenance will be easier in the future.

Impact of re-engineering

- In software re-engineering, the functionality of the software does not change. Only the design that implements functionality will change. For the end user, the software looks and (usually) feels the same.
  - The software may also become more efficient in re-engineering meaning that its limitations will be lighter. They may never become harder in the re-engineering.

Re-engineering and development

- Software re-engineering differs from development because the requirements of the re-engineered software are already known. Thus, a requirements engineering stage is not necessary.
- Moreover it is possible to simplify software design by using parts of the design plan of the old system.
  - For example, the software architecture plan of the re-engineered software does not usually change.

Re-engineering process

- A legacy system is an obsolete software that is still used but which design and implementation details are no longer documented or even not known at all.
- A legacy system is kind of over its natural life span. Nevertheless, for various reasons the system is nevertheless in use.

Evolution strategies of legacy systems

- Sooner or later a company using a legacy system must decide what to do with it:
  - The system may be abandoned.
  - The system’s life span may be further extended by the means of maintenance.
  - The system may be re-engineered.
  - The system may be replaced with a new system.
8. Software project management

- Software project management is an essential aspect of software engineering. Without proper management it is unlikely to have good results.
- Software project management is difficult because:
  - The project is intangible. Software project managers cannot see progress: workers may appear busy and yet not efficient.
  - There are numerous different software engineering processes, and it is difficult to validate whether a specific process is suitable for building a certain software.
  - Large software projects are often unique so lessons learned from previous projects may not be transferable to new projects.

Project management tasks

- Software project management is usually a project leader’s task. Most managers take responsibility for the following activities:
  - Proposal writing,
  - Project planning and scheduling,
  - Project cost estimation,
  - Project monitoring and reviews,
  - Personnel selection and evaluation,
  - Report writing and presentation.

Project planning stages

1. Establish the project constraints (time, personnel, budget).
2. Make initial assessments of the project parameters (structure and size, iterations, inspections etc.).
3. Define project milestones and deliverables.
4. Repeat stages 5-12 until the project is finished.
5. Draw up project schedule.
6. Initiate activities and personnel according to the schedule.
7. Project team works as planned.
8. Review project progress.
9. Revise estimates of project parameters when necessary.
10. Update the project schedule when necessary.
11. Renegotiate project constraints and deliverables when necessary.
12. If problems arise, check the process and when necessary correct the project plan.

Project plan

- An essential tool of project management is a project plan.
  - It is written before the project stages.
  - It is updated during the project.
  - The project plan helps to follow project progress.
  - It gives means to finish the project on schedule.
  - It gives means to notify schedule slippages as soon as possible.
  - This is the most important document of a software engineering project!

The contents of the project plan

- In the project plan, at least the following is listed:
  - the members of the project,
  - the stages of the project,
  - the tasks and task allocation of the project,
  - the schedule of the project, and
  - the risks and their counter-actions of the project.
- The project plan may also have related documentation such as control and reporting protocols.

8.1. Project schedule

- Creating a schedule for a project is one of the most difficult tasks of project management. Essentially one has to predict the future in the document since schedules from earlier project are not directly suitable for the new project.
- The more technically advanced the project is, the more probable it is that the initial estimates are too optimistic.
- In scheduling, the total work involved is divided into small activities which are estimated more or less accurately.
Milestones and deliverables

- A project has **milestones**
  - A milestone is for checking whether the project is on schedule and budget.
  - A milestone is usually at the end of a larger stage.
- and **deliverables**.
  - A deliverable is a result of the project that is relevant to the customer.

Creating the schedule

1. The project is divided into tasks.
2. The length of each task is estimated.
3. Task dependencies are figured out.
4. Tasks that have a deliverable or a milestone are figured out.
5. Tasks are combined into a **activity network**.

Activity network

- An activity network describes dependencies, order, and schedule of allocated tasks.
- Usually projects have common milestones where all active tasks are joined.
- A project may have several different activity networks.
  - Usually the project leader is responsible for the activity network.
  - The result network will depend on how to deal with risks etc.

Critical path

- Each task has
  - the earliest time when it may start,
  - the latest time it must start,
  - the earliest time it may end
  - the latest time it must end
  - schedule flexibility that defines the limits of when the task won’t be late.
- Not all tasks have flexibility. Those tasks create a **critical path**.
- If a task that is on the critical path is delayed, the whole project will be late.

Activity bar chart

- Usually an activity network is drawn as a compact **activity bar chart**.
- The activity bar chart includes:
  - all tasks and their length,
  - milestones,
  - parallel tasks, and
  - the schedule flexibility of all task and milestone end times.

Schedule example
8.2. Risk management

- A successful finish of a project is tried to ensure also when an unwanted incident occurs: a risk becomes materialized.
- In risk management
  - Risks that threaten the project are identified,
  - Identified risks are analysed,
  - Risk minimisation strategies are planned, and
  - Identified and materialized risks are monitored.

What is a risk?

- A risk is an event that
  - is possible (probability > 0 but < 1)
  - probability = 0: impossible event.
  - probability = 1: a constraint of the project.
  - when becomes materialized will harm the project.
- A risk may be
  - of the project:
    - will affect schedule and available resources.
  - of the product:
    - will affect the quality of the developed product.
  - of the company:
    - will affect the development or customer organization.

Risk management process

- All risks that may affect the success of the project should be identified.
  - In practice most improbable and meaningless risks are ignored.
- At least the following types of risks may arise:
  - technology risks,
  - people risks,
  - organisational risks,
  - tool risks,
  - requirements risks,
  - estimation risks.
Risk analysis
• Estimate the probability and seriousness of each risk:
  – probability = how certain it is that the risk will happen
    • usually a classification (e.g. five classes from very low to very high), can be a number, too.
  – seriousness = how fatal the risk is to the project
    • a classification (e.g. disastrous, serious, acceptable, small, meaningless)
• Decide how to prepare against the identified risks:
  – What risks are managed in the risk plan
  – Usually all disastrous and all serious risks with a moderate or higher probability are managed.

Risk strategies
• Each managed risk has a risk strategy = what to do if the risk becomes materialized.
• A strategy may be
  – risk avoidance:
    • lower the probability of the risk.
  – risk effect minimisation:
    • reduce the impact of the risk.
  – contingency plans
    • how to continue after the risk has happened.

Risk monitoring
• Risk monitoring involves regularly assessing each of the identified risks during the life cycle of the project.
  – In case of the risk happening the planned strategies are executed.
• The project plan may need updates:
  – New risks will arise during the project
  – The probability or seriousness of a known risk may change.
  – A risk without a planned strategy may happen.

8.3. Software cost estimations
• Software cost estimation and initial software project schedule are created at the same time.
• The largest expense of a project is the work, so in practice the expenses correlate strongly with the amount of work hours used.
• Work hour costs include management, cleaning, networking, and library services, plus office expenses, among other things.

Software costs and outsourcing
• Software development is expensive. Salaries and expenditures may easily double the expenses of a worker’s visible monthly salary.
• Due to this a lot of software companies have outsources complete software project stages, such as requirements engineering or testing.

Productivity
• In a manufacturing system, productivity is measured by dividing number of produced units by the number of person-hours required to produce them.
• In software engineering, a similar measurement would tell nothing. Something else is needed.
• Productivity estimates are usually based on measuring attributes of the software:
  1. With size-related metrics
  2. With function-related metrics
Size-based estimation

• The most common measurement of size is code.
  – A software product is program code
  – A large software has a lot of program code
  – The size of program code is easy to measure.
• However:
  – Writing code is only a part of the job.
  – The power of expression of the code depends on the chosen programming language.
  – Lines of code may be calculated only when there is something to calculate: that is, after the design stage.

Functionality-based estimation

• Instead of counting lines of code, it is possible to estimate how much functionality a software will have.
• In functionality-based estimation, it is first calculated what kind of things the software should do and then these values are translated into a functionality size estimation. The result is independent of the chosen programming language.
• A drawback of the estimation technique is that the result itself tells nothing. It is just a number.

Function point analysis

• The best known functionality measurement is the function point analysis.
• In the analysis the following values are counted:
  – number of inputs and outputs the system will have,
  – how much interaction will be in the system,
  – number of external interfaces and
  – number of files used.

Object point analysis

• A later technique than function point analysis is called object point analysis. In it the calculation is based on found functionality from the design.
• The result of the analysis is a plain number that can be normalised to express the product.
• \( OP = \sum (\text{number of objects} \times \text{weight factor}) \)

Object point counting

• Three types of entities are counted:
  – number of separate screens,
  – number of reports produced,
  – number of modules to be developed.
• The results are analysed separately and then added together. The result of the analysis is the actual number of object points.

Object point entity analysis

• The value of an entity depends on how complex its implementation is considered (weight factor):
  – Simple screen: 1op
  – Average screen: 2op
  – Difficult screen: 3op
  – Simple report: 2op
  – Average report: 5op
  – Difficult report: 8op
  – Module/component: 10op
Algorithmic cost modelling

- Algorithmic cost estimate for software cost can be expressed as:
  - Effort = A * Size^B * M, where
    - A = constant factor that depends on local organisational practices and the type of software that is developed,
    - Size = either lines of code or function/object points
    - B = constant between 1 and 1.5
    - M = multiplier made by combining process, product, and development attributes.
  - The modes are based on statistical analysis of finished projects.

COCOMO II

- COCOMO (COstructive COst MOdel) is a size-based cost estimation model that was introduced in 1981.
  - A linear process
  - Procedural programming languages
  - No reuse
  - Currently COCOMO's second edition COCOMO II (1995) is in use
    - A modern approach to process
    - Objects, reuse
    - Increasingly detailed sub-models

COCOMO II sub-models

- Application composition model
  - used when the software is implemented mostly from already finished components.
- Early design model
  - used after the requirements engineering stage but before the design stage.
- Reuse model
  - used to compute the effort required to integrate reusable components and/or program code.
- Post-architecture model
  - Used once the system architecture has been designed and we have detailed information of the developed system.

Application-composition model

- We use the Application-composition model of COCOMO II as an example of the models. Its estimation formula is based on object points:
  - PM = (NAP x (1 - %reuse/100)) / PROD.
  - PM = person-months: work estimate
  - NAP = number of object points (in COCOMO II the are called application points)
  - %reuse = how much of the code is reused
  - PROD = productivity factor (next slide)

Productivity factor

- Also the following factors must be taken into account in the Application-composition model:
  - How experienced are the members of the development team
  - How advanced development tools are in use
- The factors have the following values:

<table>
<thead>
<tr>
<th>Team experience</th>
<th>Very low</th>
<th>Low</th>
<th>Average</th>
<th>High</th>
<th>Very high</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool advance level</td>
<td>Very low</td>
<td>Low</td>
<td>Average</td>
<td>High</td>
<td>Very high</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>13</td>
<td>25</td>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

Application-composition model 2

- The final productivity factor, PROD, is the average of team experience and tool level factors:
  - PROD = (team factor + tool factor)/2
- The model is intended for component-based software estimations, but it can be used for other types of software as well.
  - It gives quite rough estimations of the needed resources.
9. Quality management

- Software quality has improved considerably in the last 20 years. There are several reasons for this:
  - modern process models support more flexible and yet controlled working habits,
  - software engineering techniques are usually mastered quite well, and also
  - software quality management has become a significant activity in software development.

Quality management tasks

- Software quality management
  - ensures that developed software fulfils its required quality level,
  - defines needed quality standards and methods necessary for ensuring required quality,
  - ensures that defined standards and methods are also followed in practice, and
  - tries to develop a suitable quality culture where quality is everyone’s responsibility.

What is software quality?

- In short, software quality implies correspondence between software and its specification. However, this is a problematic definition:
  - The customer may see quality from a different angle than the development team. Customer: usability, reliability, efficiency; Team: maintainability, reusability etc.
  - Not all quality attributes can be described unambiguously (maintainability etc.)
  - Although a software may fulfil its definitions, it perhaps does not fulfil the needs set to it by its customer. Is such a software a quality product?

Quality problems

- The previously listed quality definitions can’t be validated before the developed product is ready and customers have given feedback about it. Thus, quality can be confirmed but not verified.
- Successful quality management ensures that both the development team and the customers consider developed products to be of good quality.

Quality attributes

- The best way to defined quality is with quality attributes.
- A quality product fulfils its quality attributes well enough.
- Quality attributes have a close relationship to non-functional requirements.
  - Thus, non-functional requirements are often called quality requirements.

Software quality attributes

- Safety
- Security
- Reliability
- Efficiency
- Usability
- Learnability
- Resilience
- Robustness
- Adaptability
- Faultlessness
- Complexity
- Modularity
- Testability
- Portability
- Maintainability
- Reusability
Quality requirements

- The type of a software states what quality attributes must be considered in development and how. The software has quality requirements.
- Quality requirements are non-functional requirements that usually affect the whole system.
  - Sometimes a quality requirement may affect a single function. For instance, a response time of a function is a quality requirement.

Selecting quality attributes

- Usually quality attributes conflict with each other in a software.
  - For instance, security may conflict with usability.
- In development, it is important to decide which of the quality attributes are the most important ones and which need less attention.
  - For instance, if security and usability conflict, usability probably loses.

Selecting quality attributes 2

2. Quality planning
   1. Figure out and decide which of the quality attributes are important in a new project.
   - Select standards and methods that are the most suitable for the new project from the quality manual.
3. Quality control
   - Make sure that projects follow earlier agreed quality standards and methods.
   - Small projects may not need this kind of a very formal quality management.

Quality management activities

- The larger is an organisation and the bigger are their systems, the more important it is to have a precisely controlled quality management process.
- Quality management can be structured into three main activities:
  1. Quality assurance
     - The establishment of a framework of organisational procedures and standards that lead to high-quality software.
     - Maintains a quality manual that includes standards and methods used in the company.

Quality management activities 2

1. Figure out and decide which of the quality attributes are important in a new project.
   - Select standards and methods that are the most suitable for the new project from the quality manual.
3. Quality control
   - Make sure that projects follow earlier agreed quality standards and methods.
   - Small projects may not need this kind of a very formal quality management.

Quality management team

- It is often wise to leave quality management to a quality management team that is independent on software projects.
  - The quality management team follows ongoing projects and reports to their executive groups.
- A quality management team offers an objective independent view to ongoing projects.

Standards

- Standards are agreements whose purpose is to support consistent documents.
- A standard may be of products or of processes.
  - A product standard consists of structure and presentation presentation rules.
  - A process standard consists of process stages and the used methods of each stage.
Measuring

- Measuring is a process of calculating and collecting values of the used development process during projects.
  - Measures are used when a project or product quality and progress is controlled.
  - Measures at the end of a project are used as a history data for future projects.

Interesting and measured attributes

- The measured attributes are usually not interesting by themselves. The measured values must be interpreted.
  - the interpreted result of the measured values tells something useful about the interesting attribute.
  - For example, the measured value of number of screens found in requirements engineering can be interpreted as a usability value.

Interesting and measured attributes

Interesting:
- Maintainability
- Reliability
- Portability
- Usability

Measured:
- Length of user manual
- Number of errors
- Program size in lines of code
- Cyclomatic complexity
- Number of procedures in a set

Figure © I. Sommerville 2001

Product metrics

- Static metrics:
  - Collected by measurements of project deliverables:
    - designs,
    - code,
    - documents.
  - Can be measured from the beginning of a project.

- Dynamic metrics:
  - Collected by measurements made of program in execution.
  - The received values depend on how the program is used:
    - Usage of different functions,
    - inputs.
  - Can be measured after there is something executable.

What do metrics show?

- Static metrics are about the architecture of a product. They may have an indirect relationship to quality attributes.
  - For example, a large requirements analysis stage may imply a badly maintainable piece of software.
- Dynamic metrics are about the behaviour of a product. They usually have a direct relationship to quality attributes.
  - For example, execution time, recovery etc.

Process metrics

- Process metrics are used in process control and improvement.
  - Time metrics, such as time needed to finish a stage etc.
  - Resource metrics, such as number of used person-days, cpu-time used etc.
  - Transaction metrics, such as found errors in testing, number of change requests etc.
Metrics results analysis

- Usually several factors affect a measured attribute.
- The interpretation of results is not certain.
  - For example, if only few errors are found in testing, the reason may be in good programming, bad testing, skilful design, careful inspections, active reuse etc.

10. Summary

- Software engineering has clearly the following areas of interest:
  - project management: who does what, when, what to notify when doing it,
  - problem specification: what to do,
  - software design: how to do,
  - implementation: do it,
  - verification and validation: ensure and maintenance: look after the software.

Areas of interest and stages

- The listed areas of interest also form software engineering process stages.
- The stages may be executed in various orders but nevertheless they are present in all process models.

Project planning – don’t underestimate

- Although each stage is important, I think that the most important one is project management.
- Design does not begin from the product but from the process. Without a careful plan it is highly improbable to develop a quality product.
  - This is the stage that is especially weak in our software engineering student projects.

Realistic schedule

- Before a single requirements engineering task your project must have a clear and realistic schedule that is followed as well.
- When the schedule is realistic, working is easier, results are better and still the product is ready in time.
- Good risk management that is consistent with the schedule is very useful.

Final words

- It is always better to have a small high-quality software that can be further developed in future releases than a huge software that does not work right.
- That is the start and that is the end: good project management. The rest you can check from these slides or from the course book.