1. Introduction

- 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 very modern since 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

- 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 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 use the system via one or more subsystems (user interfaces).

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 abilities 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 somewhere 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 building was straightforward and uncontrolled. Still, even at 50's software could be 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 building 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 these two conferences defined the basis of Software Engineering.

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 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 for example sufficient for designing bridges. The model was copied directly to Software Engineering.
- Soon software engineers noticed that the waterfall model is not suitable for building any type 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.
- The first efficient programming languages Cobol and PL/1 were introduced at early 60’s.
- A remarkable step was also the C-language from 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 from languages to all Software Engineering fields.

1990’s and personal computing

- In 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.

Distributed today (2006)

- Currently (2006) 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.

Specialized today

- 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 share.
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 legacy systems.
- Legacy systems often need new functionality.
- The life span of old software is often expanded beyond natural limits.

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 see from software that it works as planned.
- Currently such negative phenomenons as spam, viruses and worms damage trust.

3. Software process

- **Software process** is 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 for different organizations and application domains.
  - The process model describes general principles, not details. Thus the process model defines frames 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 (evolution).

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:
  - Architectural design
    - Define a high-abstraction model of the system showing subsystems and used components.

Design levels 2

- Design levels continue:
  - Abstract specification
    - Define services and restrictions for each subsystem.
  - Interface design
    - Define interfaces between subsystems.
  - Component design
    - Define service components for each subsystem.
Design levels 3

- Design levels continue:
  - Data structure design
    - Define data structures that are necessary for the implementation.
  - Algorithm design
    - Define algorithms that are necessary for the implementation.
- Components may for instance be objects or object clusters.

General model of the design process

Implementation and unit testing

- In implementation, a set of components is created from the design.
  - Software component: a piece of software that has a clear interface outside and that does one thing very well (and nothing else).
  - A component may be anything 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.
  - Errors are found during software usage.
  - User requirements change or specify
  - System environment and computer equipment change.
  - Ways of using the system 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 model.
  - 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 when adding functionality.
  - Agile process models: light development.

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

1. Requirements engineering
   - what are the services, limitations, and goals of the system?
   - create models of the software that can be used to verify the meaningfulness of the requirements.

2. Design
   - specify the previous models:
     - subsystems and subsystem cooperation.
     - specify each subsystem to a required abstraction level:
       - enough abstraction to have a straightforward implementation.

Waterfall model stages 2

3. Implementation (+unit testing)
   - implementation in small pieces
   - perhaps several programming languages
   - each piece of software is tested in implementation
   - unit testing techniques

The model is strongly based on good documentation.

4. Integration and testing
   - implemented pieces are integrated to subsystems and subsystems to the software system.
   - integration testing: are interfaces functional
   - verification (build the product right) and validation (build the right product) of the system

5. Maintenance
   - after software delivery
   - repairs, new functionality

Evolutionary development

- Create software via intermediate versions:
  - Intermediate versions are prototypes or partial software that is developed stepwise to the final delivered software.
  - Will start from the best understood pieces.
  - Customers give feedback of the intermediate versions.
  - Feedback controls the development of the next version.
Evolutionary development

Advantages of evolutionary development

• Customers’ needs are well taken care:
  – It is difficult to give exact definitions to requirements.
  – When the customer sees a working prototype, he/she finds it easier to tell what he/she really wants from the system.
  – Misunderstandings are repaired earlier = less expenses.
• User interface is always visible:
  – Software learning curve should be gentle.

Disadvantages of evolutionary development

• The process is difficult to follow:
  – When is the product actually deliverable?
• Design problems:
  – Continuous changes may break the designed architecture.
• Tool support (application builder):
  – Building prototypes requires suitable tools (and skills to use them).

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 or a specific service of the software.
• The customer may find it difficult to understand why the prototype is not a delivered product.
  – The prototype is a partial implementation.
  – The prototype does not fulfil quality requirements.

Reuse-oriented process models

• 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),
  – from Free Software projects (dangerous since there is no maintenance guarantee).
• Reusable components are held in a component library.

Reuse-oriented process model stages

• The stages of processes are as follows:
  – Requirements elicitation and analysis
    • Requirements collecting and analysis.
  – Component analysis
    • Seek suitable components for the requirements from the component library.
  – 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 fulfills customers’ needs

Incremental delivery

- In the waterfall model software is specified and designed in 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 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 those requirements are processed that are relevant to it.
- The requirements of unimplemented cycles may change.

Incremental delivery strategy

- Advantages:
  - Even the first cycle creates a usable product.
  - Most important functionality is implemented first.
  - False definitions are noted and fixed in time.

Spiral development

- Each spiral models a process stage
- Four sectors:
  - problem scope = objectives and schedule
  - risk analysis = what might go wrong
  - development and verification
  - planning = result evaluation and next stage planning
- No fixed tasks
- Each spiral may be based on a different process model
- Examples:
  - 1. spiral: evolutionary development requirements engineering
  - 2. spiral: waterfall model software design
- Can be used in software maintenance as well

Spiral model example

- Determine objectives, alternatives and constraints
- Evaluate alternatives, identify critical risk areas
- Risk analysis
  - technical aspects
  - environment
  - technical support
  - human aspects
  - organizational aspects
- Plan risk responses
  - key personnel
  - technical support
  - budgetary estimates
  - development plan
  - selected risk areas
- Implement risk responses
  - personnel training
  - purchasing agreements
  - interface agreements
  - development schedule
- Evaluate risk responses: status, deviation, effectiveness
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 process.

Dawn of agile process models

- The process models supported especially the development of large long-life-span software, but they turned out to be too inflexible when building small and medium-size software.
- Due to the conflict 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 a iteration cycle is only a few weeks. In that time all stages from requirements engineering to delivery are executed.
- Due to the short cycle and 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
  - On each cycle new functionality is implemented to the software. The functionality is based on the customers’ feedback.

Agile process model principles II

- People are more important than process
  - The abilities and working style of the project members are more important than a heavily controlled process.
- Accept change
  - Requirements change so the system must be designed to support change.
- Keep it simple
  - Both the software and the process must be as simple as possible (but not any simpler).

Extreme programming (XP)

- Software is developed in very small cycles.
  - First a small core is built
  - On each cycle as small sensible set of requirements as possible is implemented on top of the core.
  - The implementation order depends on the 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. The program code is the most important document.
- The model includes a set of process rules.
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 should offer (services),
  - what affects the complete system and how (quality attributes), and
  - what limitations the system has.

Ambiguous requirements

- The term requirement is ambiguous.
- First, it implies a very general description of system functionality. Second, it implies a detailed structural description of a small detail of the system.
- Both extremes are important since the results of the requirements engineering are used at different project stages.

High-level and low-level requirements

- High-level 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 ceos and other high management staff.
- Low-level requirements are for
  - defining a contract between the software development company and the customer,
  - input to the design stage of the software.

User requirements and system requirements

- If in we do not distinct high-level and low-level requirements in requirements engineering, the abstraction level of requirements may be wrong to a specific task.
- Thus, requirements are divided to high-level user requirements and low-level system requirements.

User requirements

- User requirements are statements, in a natural language plus 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
  - systems’s operational constraints.
- The system requirements give exact definitions of what the system does and what constraints it has.

Requirements example

- LIBSYS shall keep track of all data required by copyright licensing agencies in the UK and elsewhere. (User requirement).
- LIBSYS copyright licensing; (System requirement)
  1. On making a request for a document from LIBSYS, the requestor shall be presented with a form that records details of the user and the request made.
  2. LIBSYS request forms shall be stored on the system for five years from the date of the request.
  3. All LIBSYS request forms must be indexed by user, by the name of the material requested and by the supplier of the request.
  4. LIBSYS shall maintain a log of all requests that have been made to the system.
  5. For material where authors’ lending rights apply, loan details shall be sent monthly to copyright licensing agencies that have registered with LIBSYS.

Requirements’ readers

- User and system requirements must be separated since stakeholders have several needs for the requirements.
  - The user requirements readers are usually managers and end users that are interested of general principles without difficult details.
  - The system requirements readers need details in order to design the system or find out how the system fits into the business model of the customer company.

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 II

- 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 III

- 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 are either user requirements or system requirements.
• Functional requirements may be modeled as
  – services: what does the system offer,
  – functions: what commands does the system have,
  – both: what commands implement what services (cross-checking).

Non-functional requirements

• Non-functional requirements are at least as important as functional requirements. They define who the user feels the system.
• It is possible that in case of a non-functional requirement failure the complete system is useless.
• Organizational non-functional requirements affect e.g. maintenance.

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.

Domain requirements examples

• "The deceleration of the train shall be computed as:
  \[ D_{\text{train}} = D_{\text{control}} + D_{\text{gradient}} \]
  where \( D_{\text{gradient}} \) is 9.81ms\(^2\) * compensated gradient/alpha and where the values of 9.81ms\(^2\)/alpha are known for different types of train."
• "The system uses a standard Oracle 10.X database interface via an Oracle driver."
Software requirements document

- *Software requirements document* (or software requirements specification, SRS) gives a detailed description of all software requirements and models.
- The structure of the document depends on the type of the software.
  - For example, an information system document is different from an embedded system document.

Document structure

- The structure of the requirements document must be suitable 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,
  - the basis of system testing, etc.

Document contents

1. Preface
   - Intended audience, version history, and summary of changes since the previous version.
2. Introduction
   - Overview of the system, most important functions, and cooperation with other systems.
   - May also give a short overview of the old system and references to the documentation of the old system.

Document contents 2

3. Vocabulary
   - Terminology; a reader of the document need not to be an expert of the system domain.
4. User requirements
   - Functional and non-functional requirements presented in a natural language and in diagrams and figures.
   - High-level domain requirements belong to this section.

Document contents 3

5. System architecture
   - An overview of the structure of the software.
   - Describes how services divide into subsystems and components.
   - Lists and describes already available reusable components.
6. System requirements
   - Detailed functional and non-functional requirements.
   - A standard requirements specification style for all system requirements.

Document contents 4

7. System models
   - Detailed models of sub-systems, components, and relationships.
   - System models are the basis of design. One or more modelling languages are used for them.
8. Life cycle of the system
   - The expected changes in hardware, usage, and software during the expected system usage time.
9. Appendixes
   • Known documents and citations that affect
     the system but that have not been defined in
     requirements engineering.
   • For instance, a description of the used
     database management system could be such
     an appendix document.

10. Index

4.1. Requirements engineering process

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

Requirements engineering activities
(same as slide 27)

Figure (C) I. Sommerville 2004

Requirements engineering spiral

• Also requirements engineering may benefit
from a spiral-like view.
• A spiral gives a clear implication that not all
requirements may be elicited at once but
instead several iterations are necessary.
• The requirements spiral also allows parallel
other activities such as
design/implementation during the
requirements engineering.

Requirements engineering spiral 2

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?
Feasibility study 2

- Moreover, although Sommerville does not mention it:
  4. 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 a 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,
  - the required performance of 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 — conflicts.
- Stakeholders may not be able to tell what they desire from the developed system.

Elicitation and analysis stages

- The stages of elicitation and analysis are as follows:
  - 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 each spiral is for smaller and smaller details.

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: give real-life examples, scenarios, 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 are used to define new system requirements.
- The result of the stage is a list of prioritised user and system requirements of the entire system.

Requirements validation

- The goal of requirements validation is
  - to compromise conflicting requirements
  - and to prove that the specified system is what the customer wants.
- Validation is very important because faulty requirements affect the system during its complete 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 procures 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?

Changing requirements

- The larger is the system, the more certain it is that its requirements will change.
  - The needs of stakeholders will vary.
  - New requirements are found after system delivery.
- Changing requirements must be taken into account both in the project schedule and in the developed software.
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.
  - At the highest abstraction level of design is an architectural plan. At the lowest level is pseudo code.

Nature of design

- Software design is a creative activity that should not be limited too much to processes.
  - 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 to cooperating subsystems.
- Architecture design is a stage where subsystems are identified and message passing 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, layered architectures are 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 system model descriptions,
  - the system’s division to subsystems and of each subsystem its division to components,
  - communication methods of subsystems.

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 of the offered services need not to know anything about service implementation.

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 a 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 modeling the real word 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 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:
  - “Verification: Are we building the product right?”
- In validation, it is ensured that the product fulfils its expectations from the customer:
  - “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 software inspections, a set of people analyse and check system descriptions, such as requirements documentation, system models, and program code in a formal inspection meeting. Inspections are a static technique. They do not require an executable program.
  - Software testing.
    - In testing, a software or a smaller piece of it is executed with known test 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.

Goal of verification and validation

- The goal of V&V is to ensure that the software fulfils its set needs.
  - 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 project output is inspected. The goal of the inspection is to find deficiencies and errors of the inspected output.
  - Deficiency = incomplete definition or missing functionality.
  - Error = Wrong definition or unwanted action.
Inspections as a review technique

• An inspection is the most formal review technique.
  – A review is a technique where one or more people seek deficiencies and errors from someone else’s output.
• Inspections may be used at any stage of the process. Any document may be validated with an inspection.
• Inspections (and more general: 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 check the inspected output in advance for about two hours.

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: describes the inspected subject.
  – Author/owner: represents the authors of the inspected document.
  – 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 deficiencies and errors, and
  – the inspected document has no visible deficiencies or errors.

Inspection meeting

• The inspection meeting may last the maximum of two hours. During the time the participants concentrate solely on finding deficiencies and errors.
• The participants in the inspection do not discuss the deficiencies that have been found. When a deficiency 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 deficiencies and errors must be corrected but a new inspection meeting for the document is not necessary.
  – Rejected: found deficiencies 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 cellular 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 if 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 an **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 have been tested, it is still tested that the subsystems act together correctly.
Integration testing
- In the integration testing the cooperation of the ready and separately tested functional components of the subsystem is tested.
- The integration can be top-down (it is begun with directing parts) or bottom-up (it is begun with performing parts).
- 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 such a condition that 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. Defects caused by interface errors or wrong expectations of interfaces are looked for in it.
- 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

- A delivered software is only at the beginning of its life cycle.
- During its use
  - new requirements arise,
  - old requirements change,
  - errors must be corrected, and
  - new environments require its porting.
- Pressures of change arise to software.

Pressure of change

- Pressures of change cause software evolution.
  - A process where pressures of change of a software are relieved by changing the software is called software maintenance.
- About 50-90% of all needed resources for a software are used after the delivery of the software.

Lehman's laws

- 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 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 kind of maintenance schedules. According to it the maintenance process is defined already in the development stage.
Lehman’s fourth law

- Organisational stability:
  - Over a program’s lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.
- Lehman’s fourth law states that it is not possible to speed up software evolution.
- Together the third and the fourth law state that software evolution is fairly independent of management decision. Software evolution is kind of a natural law.

Lehman’s fifth law

- Law of conservation of familiarity:
  - Over the lifetime of a system, the incremental change in each release is approximately constant.
- The law states that large changes to the software generate a lot of errors.
- Correcting the generated errors takes time; the next new version will be delayed.

Other Lehman’s laws

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

Software maintenance

- Software maintenance is a general term to a process that is used to modify a delivered software during its life span.
- Usually a maintenance team is responsible for the maintenance.
- Usually maintenance is about making relatively small changes to the software. The architecture of the software does not change. (It may change in evolution)

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,
  - how professional are 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 a software start to grow very high and still there is no guarantee for a quality result, it may be time to re-engineer the software.
- In software re-engineering part of the software is re-implemented and documented to such that its maintenance will be easier.

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 use parts of the design plan of the old system which simplifies design.
  - For example, the software architecture plan of the re-engineered software does not change.

Legacy systems

- 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

- When software projects grow, large problems arise:
  - uncontrolled growth,
  - missed deadlines,
  - bad quality,
  - budget overruns,
  - failed projects etc.
- A working project management is needed to solve the problems.

Project management tasks

- Make an offer to the customer.
- Create and maintain a project plan.
- Create and maintain project schedule.
- Cost estimation and control.
- Project control and inspections.
- Select and evaluate new employees.
- Reporting and project presentation to managers.
- Used process improvement.
Project planning

- A central tool for project planning is a project plan.
  - The plan is written before the actual stages of the project,
  - it is updated and complemented during the project.
- The project plan:
  - helps to control the progress of the project,
  - gives possibilities to finish the project on time, and
  - offers means to notice schedule slippages early.
- The project plan is the most important document of a project!

Project planning stages

1. Establish project constraints (time, personnel, budget).
2. Make initial assessments of the project parameters.
3. Define project milestones and deliverables.
4. Repeat stages 5-12 until the project is finished or cancelled.
5. Draw up project schedule.
6. Initiate activities and personnel according to the schedule.
7. Project team work 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.

The contents of the project plan

- The project plan lists:
  - 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.

Changes to the project plan

- The project plan is updated during the project.
  - Changes will certainly occur.
  - Some parts may change often (schedule, tasks etc.)
  - A loose schedule is better than a tight one.
- Updates must be straightforward
  - It is a good idea to distinguish static and dynamic parts of the document from each other.
- Project plan updates usually occur at milestones.

8.3. Project schedule

- A project has milestones and deliverables.
  - A milestone is for checking whether the project is on schedule and budget.
  - A milestone is usually at the end of a larger stage.
  - A deliverable is a result of the project that is relevant to the customer.
  - Usually a deliverable is scheduled to be ready at a milestone but not necessarily.

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 the 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 will depend on how to deal with risks etc.

Activity network 2

- Activity network includes task dependencies.
  - Some tasks can be started only after some other tasks are finished.
  - Task dependencies must be noted in the network.
- Tasks may be parallel.
  - Sometimes tasks that are independent of each other can be scheduled parallelly.
  - The amount of parallel tasks depends on resources.
  - Parallel tasks hasten the project but adds extra burden to schedule creation.

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.
- Tasks that are in the critical path may not be late, or else the project will miss its deadline.

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.

Some hints for schedule creation

- In a large project it is a good idea to create an activity bar chart for each stage.
  - The relationships between stages must of course be noted.
- A smaller project may need only one common activity bar chart.
- The size of tasks depends on the author:
  - fine grain: design and control of the schedule needs a lot of work.
  - coarse grain: schedule slippages are noticed later.
  - Smallest tasks: ~ 1-2 weeks.
  - Largest tasks: ~ 8-10 weeks.

Task lengths and dependencies

<table>
<thead>
<tr>
<th>Task</th>
<th>Duration (days)</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>8</td>
<td>T1 (M1)</td>
</tr>
<tr>
<td>T2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>15</td>
<td>T1 (M1)</td>
</tr>
<tr>
<td>T4</td>
<td>10</td>
<td>T2, T4 (M2)</td>
</tr>
<tr>
<td>T5</td>
<td>10</td>
<td>T1, T2 (M3)</td>
</tr>
<tr>
<td>T6</td>
<td>5</td>
<td>T1, T2 (M3)</td>
</tr>
<tr>
<td>T7</td>
<td>20</td>
<td>T1 (M1)</td>
</tr>
<tr>
<td>T8</td>
<td>25</td>
<td>T4 (M5)</td>
</tr>
<tr>
<td>T9</td>
<td>15</td>
<td>T3, T6 (M4)</td>
</tr>
<tr>
<td>T10</td>
<td>15</td>
<td>T5, T7 (M7)</td>
</tr>
<tr>
<td>T11</td>
<td>7</td>
<td>T9 (M6)</td>
</tr>
<tr>
<td>T12</td>
<td>10</td>
<td>T11 (M8)</td>
</tr>
</tbody>
</table>
8.4. Risk management

- A successful finish of the project is tried to ensure also in situations where something unexpected and unwanted happen.
  - Risks that threaten the project are identified.
  - The identified risks are analysed:
    - The likelihood of the risk.
    - The consequences of the risk.
  - The counter-actions of the risk are planned.
  - The risk is constantly assessed and plans for risk mitigation are revised.
  - Risk management maintenance.

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 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.

- Risk types that 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
    - either a percent value or a classification (e.g., five classes from very low to very high)
  - 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 noted in the risk plan
  - Usually all disastrous and all serious risks with a moderate or higher probability are noted.

Risk strategies

- Each noted risk receives a risk strategy = what to do if the risk happens.
- 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.

8.5. Software cost estimations

- A cost estimation is needed before the project as a part of the project offer to the customer.
- The most important cost factors:
  - Hardware and software costs,
  - travelling and training,
  - labour costs (clearly the largest cost factor):
    - salaries,
    - working environments,
    - social security costs, etc.

Problems with cost estimation

- A cost estimation must be created early because it is needed in the offer to the customer.
  - a too high estimation = no project
  - a too low estimation = budget overrun
- The total amount of work depends on the created software:
  - What kind of subsystems are needed for the system?
  - What kind of skills are required from the developers?
  - How much work – how long will it take?

Delivery and work contribution in software development work

- Work contribution can be measured with used work hours because it has a very good correlation to expenses (salaries, facilities)
- The size of delivery can be measured
  - as the size of the software product (lines of code) or
  - as the amount of functionality of the software product.
Software cost estimation techniques

- Software cost models
  - Gather information from earlier projects
  - Model relationships between different cost factors.
- Specialist estimates
  - Software and system specialists estimate the amount of needed work.
- Analogy-based estimation
  - based on earlier similar projects
- Parkinson’s law
  - estimate = the total amount of resources for this project
- Competition-based estimation

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.

Functionality-based estimation 2

- 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.
- \( OP = \sum [(number\ of\ objects) \times weight\ factor] \)
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

COCOMO II

- COCOMO (COnstructive 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: \( PM = \frac{NAP \times (1-\%\text{reuse}/100)}{PROD} \).
  - \( PM \) = person-months: work estimate
  - \( NAP \) = number of object points (in COCOMO II the are called application points)
  - \( \%\text{reuse} \) = 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></td>
<td>4</td>
<td>7</td>
<td>13</td>
<td>25</td>
<td>50</td>
</tr>
</tbody>
</table>

Application-composition model 2

- The final productivity factor, \( PROD \), is the average of team experience and tool level factors:
  - \( PROD = (\text{team factor} + \text{tool factor})/2 \)
- 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,
  - modern 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 fulfills 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 fulfill its definitions, it perhaps does not fulfill 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 fulfills 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.

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.
  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 2

1. Quality assurance

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.
   2. 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:
- Number of procedures per unit
- Cyclomatic complexity
- Program size in lines of code
- Number of error messages
- Length of user manual

Figure © I. Sommerville 2004

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, skillful 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 maintain: 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.

**Final words**

- Realism is the key word in schedules. It is a thousand times better to have a small high-quality software than a huge software that is about 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.