Software eng. - Introduction to SE

Teodora Ristic

November 2021

1 Introduction

Software engineering is the study or practice of using computers and computing technology to solve real-world problems. Computer scientists study the structure, interactions and theory of computers and their functions. Software engineering is a part of computer science in that software engineers use the results of studies to build tools and techniques to meet the needs of customers.

Software product consists of:

1. **Computer programs (software)** which enable user to reach a wanted effect.
2. **Data structures** which enable software to manipulate data
3. **Documents** which describe the software from implementation and user perspective.

A software product is successful if the development is completed, if the product is useful, usable and used.

1.1 Reasons for failure X

1. quality issues - the system is faulty (bugs)
2. development doesn't meet deadlines
3. over budget
4. doesn't fulfill user needs
5. it's difficult to maintain
6. inadequate development planning (no milestones defined)
7. undefined project goals
8. poor understanding of user requirements
9. insufficient quality control (testing)
10. technical incompetence of developers
11. poor understanding of cost and effort by both developer and user

1.2 Requirements to succeed ✓ 6

1. project sponsorship at executive level
2. strong project management
3. right team players
4. good decision making structure
5. good communication
6. team members working toward common goals

1.3 Characteristics of good SW products ✓

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

1.4 Engineering principles X 8

1. Evaluation of costs
2. Time planning
3. Inclusion of users when defining requirements
4. Defining development phases
5. Defining end products
6. Following and managing the project progress
7. Designing of quality control
8. Exhaustive testing

1.5 Software project actors 8 ✓

1. Project sponsor
2. Project manager
3. Project team
4. Supporting personnel
5. Customers
6. Users
7. Suppliers
8. Opponents

1.6 SW project manager challenges ✓6

1. **considering users** - poorly defined requirements; must be focused to simplicity of use and responsiveness
2. **considering the development team** - focused on development principles, technical solution
3. **considering executive management** - focused on economic success; costs and time limitations.
4. **heterogeneity** - development techniques for different platforms and environments
5. **development speed** - techniques that enable fast software development and early delivery to users
6. **trust** - techniques worthy $\overset{\text{of users' trust}}{\text{to trust}}$ by the users.

1.7 Ethics

Software engineering involves wider responsibilities than just application of technical skills:

1. **Confidentiality -** Respect for confidentiality and protection data of employers and clients.
2. **Competence -** Engineers should not misrepresent their level of competence.They should not knowingly accept work which is outwith their competence.
3. **Intellectual property rights** - Engineers should be aware of local laws governing the use of intellectual property such as patents, copyright, etc. They should be careful to ensure that the intellectual property of employers and clients is protected.
4. **Computer misuse** - Software engineers should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).

Software Process

teodoraristic

October 2021

1 Software Process

Software process is a process of building a software product from detected need until the end of maintenance. In software process we face technical and organizational challenges. The result of a software process is a software product. Different software products require different software processes. Good software process leads to a good software product, reduces risks and it's more manageable and clear.

1.1 Structure of a Software Process

ispit pitanje?

A software process consists of steps that need to be performed in correct order. Each step has to have clearly defined goals, it needs people with specific skills, it's based on clear input and output parameters, it has a defined beginning and end time, it uses specific techniques, tools, guidelines, agreements, standards.

Each step ends with its **verification** (check consistency of outputs with input parameters) and **validation** (check consistency with user needs).

2 Software Process Phases 9

2.1 Problem definition

It answers the questions: What is the problem? Who has that problem and where? Why is there a problem? Can it be further explored? If the problem exists, it has to be solved using available resources.

2.2 Feasibility study

It answers the questions: Is the project technically, economically and operationally feasible? Which are the advantages and the extent of the project? Feasibility study enables decision to start the project or not.

1. **Technical feasibility** - for each alternative implementation, we have to consider technical details for all the phases and risk needs to be estimated. Technical personnel has to be included (people part of the project team).
2. **Economic feasibility** - It takes into account costs (one time cost of equipment - hardware and software, education, consultations, support), salaries, maintenance.. It also looks into benefits - savings (salary savings, material, increase of production, usage costs) and contributions (better customer service, improved management capabilities, control of production, reduction of errors and failures)

The result of the feasibility study is a feasibility report given to the management and users. They select one of the alternatives if they want to start the project based on this.

2.3 Requirements analysis

It describes the system from user perspective in detail. **Functional requirements** show use cases for all interactions of users with the system. **Non-functional requirements** take into account operational requirements (security, safety, usefulness, performance) and evolution requirements (testing ability, maintenance ability, upgradeability). Requirements analysis is done by a **system analyst**.

The process: 6

- Obtain knowledge from the field of use

- Get acquainted with the problem through questionnaires, interviews
- Study the existing systems
- Analysis done from outside in (from outputs to inputs)
- System description done textually, with diagrams
- Revise the requirements analysis with users

The result is a document called **Software requirements specifications** which informs the users of the system so that they can estimate if the system is addressing their requirements, ie if the developer's understanding of the problem is correct. It splits the problem into smaller components and this document is later used for system design and validation.

~~The document typically includes: introduction, general description (relations to other systems, function and component overview, user specifics, limitations), description of requirements for each component, external interface requirements (hw), required performance, design limitations...~~

Requirements analysis ends with a review:

1. Review of all requirements
   - **validity** - do system functions comply with user needs?
   - **consistency** - are all the functions the user needs considered?
   - **reality** - can the requirements be satisfied with the system and within cost limitations?
2. Review of individual requirements
   - **testability** - can the fullfilment of requirements be checked?
   - **understandability** - is it clear why these requirements exist? are the requirements correctly understood?
   - **traceability** - is it clear why these requirements exist?
   - **adaptability** - can the requirement be changed without influence on other requirements?

2.4 System design

structure of components + external properties of components + relationships between components

In system design, system architecture is defined: structure of software components, external properties of each component and component relationships. Here whole system requirements are divided into requirements of individual components. Division of a system into components is called **decomposition**. We have to make sure that the components are as independent as possible which is important for maintenance.

System components include processing and data components. Decomposition is vertical and horizontal:

1. **partitions** - vertical division into multiple weakly connected subsystems, which offer services at the same level of abstraction.
2. **layers** - horizontal division into subsystems where higher layers use services of lower layers and offer services to higher layers.

2.5 Component design

component structure + modules that need to be implemented + database and file formats

Based on selected system design, each individual component is further designed. We differentiate between component structure, modules that need to be implemented, database and file formats. Each component is defined with such level of detail that it allows component implementation.

Products of component design are: component specifications (diagrams, pseudocode, descriptions), data file design (organization, access principles), hardware specification, testing plan, time plan for implementation. The phase ends with a technical review. (component design)

Products of component design:
1. component specifications (diagrams, pseudocode, descriptions)
2. data file design (organization, design principles)
3. hardware specification
4. testing plan
5. time plan for implementation

2.6 Implementation

-> ends with verification

In the implementation phase we build the software product that is made up of:

1. a collection of components that can be developed for a specific purpose, obtained from somewhere else, modify existing components
2. data structures that enable software to process data
3. documentation that describes operation and usage of the software

Implementation ends with verification. Individual components are tested to see if they comply with component specifications. They are integrated into a software system and tested against the system specifications.

2.7 Testing

-> ends with validation; documentation is finalized

In the testing phase, software is validated (check to see if it complies with the user needs). It's executed using prepared testing plan that follows SRS. After correcting problems, the testing phase has to be repeated from the beginning. During testing, the documentation is finalized.

2.8 Deployment

In this phase the product is deployed to users that can accept or reject it.

2.9 Maintenance

The system has to fulfill user needs correctly and constantly. Maintenance is changing the product if there needs to be some **corrective changes** (error correction), **adaptive changes** (extension of functionality, modification of interfaces), **perfective changes** (make the product better, transfer to other platforms), **preventive changes** (work against software deterioration).

The software cycle ends when the software is not used anymore.

3 Models of software development process

6

Model of a software development process is a plan of executing phases in a software process. Each process includes all the phases, either implemented formally or informally and in different order.

Advantages of formal processes: 8 X

- better control over resources
- better relations with the customer
- shorter development time
- lower costs
- higher quality and reliability
- higher productivity
- better work organization and coordination
- less stressful work.

3.1 Build and fix model

Its difficulties: missing specification and design plan, there are no defined phases, no milestones. It's an inconvenient approach because it's impossible to control, it's difficult to assure quality, high load on the customer causes less trust in the product.

$$\begin{tikzcd}
\text{build 1st version} \arrow[d] \\
\text{modify until client is satisfied} \arrow[loop right] \arrow[d] \\
\text{maintenance phase} \arrow[d] \\
\text{retirement}
\end{tikzcd}$$

3.2 Linear model (classic, cascade, waterfall)

The phases follow one after the other, without overlapping. Each phase ends with validation and verification. Using it is possible only if there are well-defined requirements and project goals, because later modifications aren't possible. It requires patience from the customer. The working software is available only after deployment at the end of the project.

Advantages: clarity, separation of phases, permanent quality control, good control over expenses. In practice, the next phase enables better understanding of the previous one and revisions of previous phases are needed. Linear model doesn't allow those, though.

Linear model with corrections allows revisions of previous phases. Partial overlap of phases is possible.

3.3 V model

It's an extension of the linear model. Each phase of development is related to some phase of testing.

3.4 Spiral model

Each cycle includes phases of the linear model. In each cycle a prototype is developed and it includes four quadrants:

X

4 quadrants

1. definition of goals, alternatives, limitations - **goals** (functionality, performance, interface definition, success factors); **alternatives** (new development, component reuse, buying a component or product, subcontracts); **limitations** (costs, time limits, interfaces).
2. evaluation of alternatives, risk identification - evaluation of alternatives according to goals and limitations; risk identification (lack of experience, new technologies, short time limits); risk evaluation (failure probability, the sense of further development. . .)
3. development of a current cycle product - the cycles of the linear model: design, implementation, testing; quality control (reviews, verification, validation)
4. planning the next cycle - project planning of the next cycle; configuration management (identification, organization and control of changes); test planning, planning of system installations

Advantages of the spiral model:

- early identification of obstacles with reduced cost
- users get early insight into the system through prototypes
- development of critical components first
- acceptable incomplete design and planning
- early feedback from users

Limitations:

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

Use it when:

- prototypes are acceptable
- risk management is crucial, projects with higher risks
- longer development time period is acceptable
- in the case of complex requirements
- changes of requirements are expected

3.5 Unified Process model

Unified Process (UP) is a modern process model framework that can be adapted to meet needs of specific organization or projects. It defines development phases and engineering disciplines to perform them. Each phase is built of steps, which are the basic elements of a process.

A project consists of 4 phases:

1. **Inception -** establish the business case for the system; it ends with the Lifecycle Objective Milestone
2. **Elaboration** - develop an understanding of the problem domain and system architecture; it ends with the Lifecycle Architecture Milestone
3. **Construction** - system design, programming and testing; it ends with the Initial Operational Capability Milestone
4. **Transition** - deploy the system; it ends with the Product Release Milestone

Unified Process defines 6 engineering disciplines: ✓

1. Business modeling
2. Requirements discipline
3. Analysis and design
4. Implementation
5. Test discipline
6. Deployment

Process is built from basic building blocks defining what needs to be made, which competences are needed and how to reach the goal:

1. **Roles (who?)**: knowledge, competences, responsibilities
2. **Products (what?)**: the result of a task - documents, models, data, programs
3. **Task (how?)**: description of a process to reach the goal

X UP good practice: develop software iteratively, manage requirements, use component-based architectures, visually model software, verify software quality, control changes to software.

4 Prototypes

4.1 Using prototypes

Using prototypes for software development is a way of rapid development. Original purpose of prototypes was to help the customer and developers to understand the requirements better (users can check how their requirements reflect the system; prototype reveals errors and missing requirements).

→ Prototype development is an activity of reducing risks at requirements analysis. Prototype is available early and it can be used to educate users and test systems.

4.2 Prototype development

Historically a prototype was considered inferior and so the prototype development was always followed by further development. Today there are two options:

1. **Evolutionary development** - an initial prototype is improved and leads to the final software system. The purpose of this prototype is to build the final system. Development starts at the most clear requirements.
2. **Development with prototype omission** - prototype serves only to clarify requirements and it's discarded later. The final system is developed independently, based on clear requirements. Development starts with most unclear requirements.

4.3 Evolutionary development

Evolutionary development is needed when specifications can't be given before developing the system (prototype). Verification isn't possible because there are no specifications. It can only be made as a demonstration of system capabilities. Evolution development is based on rapid prototyping techniques.

Prototype can sometimes serve as system specifications, but not completely because some requirements can't be included in prototypes (security critical functions), legally prototypes can't be a contract agreement, nonfunctional requirements can't be verified on prototypes (security, testability, maintainability, upgradability).

Advantages:

- early delivery - sometimes it's more important than functionality or long term maintenance
- user inclusiveness increases the probability that the system satisfies user needs and the probability of the system being used

Disadvantages:

- project management usually expects linear models
- prototype development without specifications requires additional developer abilities
- maintenance difficulties - constant changes have negative influence on the system structure, making long term maintenance difficult
- contractual difficulties - users/customers must accept additional obligations, multiple deliveries for deployment

4.4 Throwaway prototyping

A prototype is developed based on initial specifications for experimenting (different properties, technologies) and then it's abandoned. It's not meant to be used in the final system. Prototypes aren't developed considering all the required system properties, they aren't documented which is difficult for maintenance, architecture of a prototype is developed ad-hoc which makes maintenance difficult too.

4.5 Phased development

→model for software developement process

System is developed and delivered in multiple phases. The basis for development of the first phase is an **architectural design** of the whole system. Phases that are deployed to users enable some incomplete functionality. Although some of it is missing, users can use it while others are developed. It has advantages over prototyping (evolution process) because it assures better project clarity, control, management and better final architectural design.

5 Software process frameworks

An organization can build its own software process framework to define its processes. Each process consists of steps: 5

- define a phase
- require some engineering discipline
- define roles of developers involved in the process
- result in predefined products
- are supplemented with guidelines to efficiently execute the process

} X

6 Software process maturity level

X

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

1. **initial** - processes depend only on individuals
2. **repeatable** - established basic project management policies; experience may be used for similar projects

3. **defined** - documentation of the standard guidelines and procedure takes place; each project follows predefined processes
4. **managed** - quantitative quality goals are set for the organization for software products and processes
5. **optimizing** - continuous process improvement in the organization using quantitative feedback

Diagramming techniques

teodoraristic

October 2021

1 Diagrams

We use diagrams as basic tools for modeling because they enable description of complex systems; unambiguous presentation allows easier understanding of a problem and solution; they make modeling of system structure and behavior easier.

Diagram is a partial graphic presentation of a system model. Model can include multiple diagrams and documentation (eg. use-case model consists of a use-case diagram and a document describing it).

Diagrams fro software development: flowchart, data flow diagram (DFD), entity-relationship diagram (ER), unified modeling language (UML) diagrams.

Diagram and model types:

1. **context -** description of a system from the outside-in its environment
2. **behavioral -** system behavior: use-cases, process behavior, data flow, state transitions
3. **structural -** structure of a system, components, objects, data

Diagramming domains:

1. **application domain -** for requirement analysis; system environment
2. **solution domain -** system design, component/object design; system development technologies

Model levels:

1. **conceptual model -** basic system properties, high-level, broad view of the system
2. **logical model -** supplements the conceptual model with details, limited to system operation and usage
3. **physical model -** adds implementation details to the logical model (types, variables, functions)

2 Flowchart

Diagram consists of symbols for start, end, control flow, process steps, I/O operations, decisions...

3 Data Flow Diagram DFD

It models data flow, not control flow. It gives logical overview of the system, not physical. It's convenient for communication with users, management and information system experts. It's suitable for analysis of existing and proposed systems. DFD shows how data is processed in the sense of inputs, processes and outputs.

4 Entity Relationship Diagrams ERD

ER diagrams are intended for illustration of a data structure.

1. **Entity** – a real world object, which can be distinguished form other objects. Entity consists of a set of attributes.
2. **Entity set** – a set of similar entities, with the same set of attributes

3. **Attribute** – a characteristic of an entity
4. **Relationship** – relation between two or more entities
5. **Key** – an attribute that enables differentiation between entities

ER diagram is mapped to a relational model, which defines how data is stored in a relational database.

5 UML diagrams

UML (The Unified Modeling Language) is used for specification, visualization, modification, construction and documentation of processes and objects during system development. The focus is on object oriented approach. It is a set of standardized system representations.

5.1 Structure diagrams 6

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

5.2 Behavior diagrams 7

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

5.3 Class diagram: domain name

Domain model is a simple class diagram intended for quick initial description of a system in the phase of analysis. It only includes generalizations and aggregations. It usually doesn't include attributes and operations (only the most important ones).

5.4 UML package diagram ✓

It's used to simplify complex class diagrams where classes are grouped into **packages**. A package is a collection of logically related UML elements (typically classes). Packages are connected by their dependencies. A package is dependent on another package if a change in one requires a change in the other.

5.5 Object diagram

Obejct diagrams are suitable for depicting class usage. They show real world objects and their relations. They're needed when class diagrams are too abstract. Object diagrams show example of class usage in one time during software operation.

5.6 Component diagram

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

5.7 Deployment diagram

Deployment diagram shows physical relations between system hardware and software. Hardware elements are computers (servers, clients), embedded devices, other devices (sensors). Deployment diagram shows hardware used by software.

5.8 State machine diagram

Some objects act differently depending on the state in which they're in. An object state depends on previous activities and changes due to external events. A state can be associated with a characteristic activity that executes at transition to this state.

The state machine diagram depicts possible object states and transitions between the states, events that influence state changes, and activities related to states and transitions.

5.9 Sequence diagram

The sequence diagrams show communication between objects as a sequence of messages. Each object has its own lane that corresponds to time of object existence and labels object activities. It;s suitable for conceptual and logical models at different levels of abstraction.

5.10 Use case diagram

The use case diagram provides an external view on the system and depicts user requirements. It shows interaction between users and the system. It defines the system boundary (what's included in the system, what's not).

Software eng. - Requirements Analysis

teodoraristic

November 2021

1 Introduction

Requirements Analysis (System analysis) is performed by a system analyst. Its goal is to define clear explicit requirements. The input information is a real problem description, which is typically poorly defined, unclear and contradictory. The challenge is to define a system that meets user needs and expectations.

Implicit models aren't verbalised, they are changing with time, they can't be expressed formally. Clients and a system analyst don't speak the same language.

The result of analysis is a **System Requirements Specification (SRS)**, a complete description of system behavior that includes functional requirements (use cases for all interactions of users with the system) and nonfunctional requirements (requirements that limit system design or implementation, eg. performance, quality standards). SRS is needed to:

1. familiarize with user needs and limitations
2. reach similar understanding of the problem and solution for analyst and user
3. define the basis for system design
4. define the basis for system validation.

The procedure:

1. getting familiar with the problem and collecting requirements
2. analysis, modeling, coordination
3. recension (validation of requirements)
4. requirements management

Nonfunctional requirements can be divided into: **operational requirements** (security, safety, usability) and **evolutionary requirements** (testing ability, maintenance ability, extensibility, upgradability).

2 Getting familiar with the problem and collecting requirements

Requirement collection may look simple, but it's not. The questions we need to ask customers and users are: What is the purpose of the system? What are the goals? How does the system fulfill the business requirements? How will the system be daily used? More difficult questions are: How to define the system scope? How to correctly understand requirements? What to do if the requirements change in time?

2.1 The system scope

The scope may not be clear: Who will directly use the system? Which other systems will be connected with this system? Which functionality should be included and which shouldn't? The customer may provide technical details that may not be required or correct and that may make understanding of the system difficult.

2.2 Requirements understanding and changes

The customers may not know the limitations of their hardware, software or processes well. They don't have full understanding of the problem domain. They can't define requirements to system analyst. The leave out information that seems obvious to them. They list opposing requirements. They list unclear requirements and those that can't be verified.

System requirements change over time because of:

1. Organizational structure changes (new products, new departments)
2. Changes in operational extent (increased number of daily transactions, higher number of users, higher communication throughput)
3. External changes - changing of law, changes of international standards
4. Customers get new ideas when they become familiar with the capabilities of the system

2.3 Information gathering steps

1. **Study of literature**:
   - Books from the problem domain
   - Process description
   - Process improvement studies
   - Development documentation
   - Error and system logs, bug tracking
   - Proposals

   6 x

   ✓ ✓
   ✓ ✓
   ✓ ✓
   ✓ ✓-
   ✓ ✓
   ✓- ✓

2. **Expert advice** - listen to experts from the problem domain. Experts aren't only those linked to the customer. The advantages of experts is that they know the problem domain in detail; they can help to define the scope, they can direct system analyst to alternative solutions.
3. **Questionnaires** - System analyst asks the customers and/or users what they expect from the new system, expecting that they're not too biased or limited by their limitations of understanding. In addition to questionnaires, interviews and brainstorming sessions are possible. Communication can be direct (oral) or indirect (written).
4. **Derivation from existing systems** - The analysis is started based on an existing system (the system to be replaced, similar system in another organization, a system described in literature).
5. **Synthesis from environment** - Software can be successful only if the properties of the system environment are considered. This is called **process analysis or normative analysis.**
6. **Using prototypes** - It's in use when requirements can't be clearly defined. Prototype is a working model of a part of a system or of a whole system. Prototypes differ from the final system because there's no compliance with nonfunctional requirements.
7. **Atomic requirements** - Requirements are atomic if they precisely define the need without being divided into more requirements; they're measurable, testable and traceable.

3 Analysis, modeling, coordination

When the requirements are collected, the following steps have to be made:

1. **Analysis** - requirements analysis in a sense of consistency, completeness and reality.
2. **Modeling** - building an explicit model of requirements in a form of neutral language, diagrams and formal notations (math equations, pseudo code).
3. **Coordination** - the proposed solutions must be coordinated with stakeholders regarding the scope, contradicting requirements, priorities and risks.

The steps repeat multiple times.

3.1 Analysis 6

✓✓✓✓✓ = ✓✓✓✓✓

1. Categorization of requirements - grouping request into related subsets
2. Finding relations between requirements
3. Testing requirements consistency
4. Identifying missing data
5. Checking requirements ambiguities
6. Defining priorities based on user needs - priorities can serve for defining the incremental or phased development

3.2 Modeling

Requirements can be presented with different kinds of models:

1. **Natural languages** - easy to understand; there are risks for ambiguity and unclarity
2. **Mid-formal notations** - more precise than natural languages, simple to understand, don't require strictly defined notations
3. **Formal notations** - eg. math expressions; they're exact, unambiguous

Requirements specification models must offer understanding of the system from 3 perspectives:

1. **Data - data model** - what form of data will be used and how different pieces of data are related
2. **Processes - process model** - which processes form a system and what their function is
3. **Execution - control model** - the sequence of executing process activities, decisions, merges, forks and joins of control flow.

There are many different approaches to analysis and modeling including: structural analysis (top down approach)
UML diagrams (object oriented approach)

3.3 Requirements modeling

Structural analysis: 6

✓✓✓✓✓ - ✓

1. list of events
2. contextual diagram
3. data flow diagrams and specifications (DFD)
4. entity relationship diagrams (ERD)
5. activity diagrams in a form of a flowchart
6. ELH

Using UML (OOA):

1. use cases (diagram+description)
2. domain diagram (class diagram)
3. sequence diagrams, communication diagrams → inter-class
4. activity diagrams, state diagrams → intra-class

3.4 Data model and procedure

Structural analysis typically uses:

1. Data dictionary
2. Entity relational diagram (ER)
3. Relational data model - typically refined later in a system or component design phase
4. Other models can also me used: eg. Entity life history (ELH)

+ pic from sys3

3.5 Object Oriented Analysis (OOA)

More strict differentiation between analysis and design. The basis is the usage of objects and presentation using object models - typically using UML.

3.6 Use cases

Use cases describe interaction between actors and a system for specific scenario of system usage:

1. Describe a service offered to user
2. Describe interaction of user and system in details, all the processes involved from the beginning till the end of usage
3. With a use-case users reach some goal
4. Understanding of requirements is crucial for defining cases
5. Use cases are defined for each system vent
6. Described in details using **use-case descriptions** and illustrated using **use-case diagrams**

3.7 Use case models

A use case model consists of **a use case diagram** and **use case description** (for each use case in the diagram. Use case models don't enable complete system description. They have to be complemented with additional requirements: data and nonfunctional requirements; physical layer requirements.

Use case description is specification of interaction between the software product and actors. It provides information on activities of actor and activities of the product of all possible interaction courses.

Use case description contents:

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

3.8 Additional diagrams and models

- ~~Use case flows can be documented using activity diagrams.~~
- ~~System processes can be presented using DFD or communication diagrams or sequence diagrams.~~
- ~~System data can be presented using ERD or class diagrams.~~

3.9 System decomposition

OOA performs system decomposition as division in components, subcomponents and objects. Structured approaches divide functionality (functional decomposition). Objects are defined with classes - selection of objects is based on domain model (simple model without attributes and operations).

3.10 Modeling of classes

The main tool for defining classes are use cases. One of the methods for defining classes is **noun identification** (it's simple, but it could be misleading). Not all nouns are appropriate for classes: different nouns can have identical meaning; nouns aren't used only for classes, but also for attributes; some nouns don't represent object of interest and they aren't needed. Search for the simplest set of classes that satisfy the requirements and offer good enough adaptability (considering maintenance).

3.11 Dynamic models

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

3.12 UML interaction diagrams

They describe the cooperation between system components (objects). They typically show behavior for individual use case by showing objects and messages that they exchange. For us the most useful among interaction diagrams are sequence diagrams and communication diagrams. They provide the same information with different emphasis.

3.13 UML Activity diagram

UML activity diagram shows activity that is typically executed inside one class method, but could also be used for multiple components or a whole system. One activity can describe multiple use cases.

3.14 Requirements coordination

After modeling, users may place higher importance on different requirements. They often want more than what can be achieved (due to cost). Different stakeholders provide different requirements that may be incompatible or contradictory. System analyst has to resolve these problems with negotiations:

1. define system priorities
2. identify risks related to requirements
3. briefly estimate required development cost for each of the requirements
4. discuss requirements supplemented with data with stakeholders.

Coordination is an iterative process of modifying, grouping, omitting requirements to reach an agreement.

3.15 The SRS document

Possible contents of SRS:

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

4 Recension

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

4.1 SRS recension (same as lesson 2)

Review of system requirements (all):

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

Review of individual requirements:

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

4.2 Recension types

1. **Control from an individual** - often performed by the author using **control lists**
2. **Group review** - by a group of reviews. The procedure isn't predefined and the roles aren't defined. Usually each reviewer makes a review as an individual using control list and then they combine the results.
3. **Supervision** - formal recension by a group of trained supervisors with defined roles. They also use control lists.

4.3 SRS supervision

The goal is to find as many errors and risks as possible, not to judge the author. Supervision doesn't correct errors, it's expensive and long-lasting. It's also effective.

4.4 Conclusion of a supervision

The moderator concludes that errors exist and need to be corrected or no errors are found. Author corrects the errors found by supervisors. If there are larger changes to be made or if there are additional errors, supervision needs to be repeated.

5 Requirements management

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

Changes of requirements are unavoidable and happen during the whole life cycle. They include identification, tracing, prioritizing, validating and communicating with stakeholders.