2011

download.benjaminsommer.com

Benjamin Sommer

[SOFTWARE ENGINEERING LECTURE NOTES]

Overview

Software Engineering is concerned with

- Technical processes of software development
- Software project management
- Development of tools, methods and theories to support software production
- Getting results of the required quality within the schedule and budget
- Often involves making compromises
- Often adopt a systematic and organized approach
- Less formal development is particularly appropriate for the development of web-based systems

Software Engineering is important because

- Individuals and society rely on advanced software systems
- Produce reliable and trustworthy systems economically and quickly
- Cheaper in the long run to use software engineering methods and techniques for software systems

Fundamental activities being common to all software processes:

- Software specification: customers and engineers define software that is to be produced and the constraints on its operation
- Software development: software is designed and programmed
- Software validation: software is checked to ensure that it is what the customer requires
- Software evolution: software is modified to reflect changing customer and market requirements

Software Engineering is related to computer science and systems engineering:

- Computer science
  - Concerned with theories and methods
- Software Engineering
  - Practical problems of producing software
- Systems engineering
  - Aspects of development and evolution of complex systems
  - Specifying the system, defining its overall architecture, integrating the different parts to create the finished system

General issues that affect many different types of software:

- Heterogeneity
  - Operate as distributed systems across networks
  - Running on general-purpose computers and mobile phones
  - Integrate new software with older legacy systems written in different programming languages

    - Challenge: build dependable software that is flexible enough to cope with heterogeneity
- Business and social change
    - Change existing software and rapidly develop new software
    - Traditional software engineering techniques are time consuming
    - Goal: reduce time to adapt to changing needs
- Security and trust
    - Software is intertwined with all aspects of our lives
    - See remote software systems (web page, web service interface)
    - Make sure malicious users cannot attack software and information security is maintained

Essential attributes of good software

- Maintainability
    - Evolve to meet the changing needs of customers
    - Software change is inevitable (see changing business environment)
- Dependability and security
    - Includes reliability, security and safety
    - Should not cause physical or economic damage in case of system failure
    - Take special care for malicious users
- Efficiency
    - Includes responsiveness, processing time, memory utilization
    - Care about memory and processor cycles
- Acceptability
    - Acceptable to the type of users for which it is designed
    - Includes understandable, usable and compatible with other systems

Application types

- Stand-alone applications
- Interactive transaction-based applications
- Embedded control systems
- Batch processing systems
- Entertainment systems
- Systems for modeling and simulation
- Data collection systems
- Systems of systems

Software engineering ethics:

- Confidentiality: respect confidentiality or employers and clients (whether or not a formal confidentiality agreement has been signed)

- Competence: do not misrepresent your level of competence (never accept work being outside of your competence)
- Intellectual property rights: be aware of local laws governing the use of intellectual property such as patents and copyright
- Computer misuse: do not use technical skills to misuse other people's computers

Software processes

Main software processes

- Specification
- Design and implementation
- Validation
- Evolution

Software process models

- Waterfall model
- Incremental development
- Reuse-oriented software engineering

Waterfall model:

- Requirements analysis and definition
    - Consult system users to establish system's services, constrains and goals
    - They are then defined in detail and serve as a system specification
- System and software design
    - Establish an overall system architecture to allocate the requirements to either hardware or software systems
    - Involves identifying and describing the fundamental software system abstractions and their relationships
- Implementation and unit testing
    - Integrate and test individual program units or programs into complete systems
    - Ensure the software requirements has been met
    - Software system is delivered to the customer (after testing)
- Operation and maintenance
    - Normally the longest life cycle phase
    - System is installed and put into practical use
    - Involves correcting errors, improving the implementation of system units and enhancing the system's services as new requirements are discovered

Incremental development:

Idea

- Develop and initial implementation
- Expose this to user comment
- Evolve this through several versions until an adequate system has been developed

Characteristics

- Start with an outline description of the software to develop
- Specification, development and validation activities are interleaved (concurrent activities) rather than separate
- More rapid feedback across activities
- Fundamental part of agile approaches
- Better for most business, e-commerce and personal systems than waterfall approaches

Benefits compared to waterfall model

- Reduced cost of accommodating changing customer requirements
- Easier to get customer feedback on the development work that has been done
- Possibly more rapid delivery and deployment of useful software to the customer. Customers use and gain value from the software earlier than with water model

Problems

- Process is not visible
    - How to measure progress (see managers)
    - Producing documents that reflect every version of the system is not cost-effective
- System structure tends to degrade as new increments are added
    - Time and money is required on refactoring to improve the software
    - Regular changes tend to corrupt its structure
    - Further software changes become increasingly difficult and costly

Reuse-oriented software engineering

- Requirements specification
- Component analysis
    - Search for components to implement this specification
    - Usually exact matches are not found
- Requirements modification
    - Analyze requirements using information about the found components
    - They are modified to reflect the available components
    - Restart with component analysis in case found components cannot be modified
- System design with reuse
    - Design the framework of the system or reuse an existing framework
    - Some new software may have to be designed if reusable components are not available
- Development and integration

  - Develop software that cannot be externally procured
  - Integrate components and COTS (commercial off-the-shelf systems)
  - System integration may be part of development process rather than a separate activity
- System validation

Types of software components to be used in a reuse-oriented process:

- Web services
  - Developed according to service standards
  - Available for remote invocation
- Collections of objects in a package (e.g. from .NET or J2EE)
- Stand-alone software systems that are configured for use in a particular environment

Advantages

- Reduce the amount of software to be developed
- Reduce cost and risks
- Usually leads to faster delivery of the software

Disadvantages

- May lead to a system that does not meet the real needs of users (requirements compromises are inevitable)
- Lost control over the system evolution as new versions of the reusable components are not under the control of the organization using them

Process activities

Software specification

Definition: the process of understanding and defining what services are required from the system and identifying the constraints on the system's operation and development

Requirements engineering: critical stage of the software process as errors at this stage inevitably lead to later problems in the system design and implementation

Requirements engineering process

- Feasibility study
  - Estimate whether the identified user needs may be satisfied using current software and hardware technologies
  - Considerations: cost-effective, development within existing budgetary constraints
  - Should be relatively cheap and quick
- Requirements elicitation and analysis
  - Derive the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis

- Development of one or more system models and prototypes possible
- Help to understand the system to be specified

- Requirements specification
    - Activity of translating the information gathered during the analysis activity into a document that defines a set of requirements
    - User requirements: abstract statements of the system requirements for the customer and end-user of the system
    - System requirements: more detailed description of the functionality to be provided
- Requirements validation
    - Checks the requirements for realism, consistency and completeness
    - Errors are discovered to correct the specification

These process are usually interleaved (see agile methods)

Software design and implementation

Software implementation: process of converting a system specification into an executable system

Characteristics:

- Always involves processes of software design and programming
- May refine software specification if incremental approach is used

Software design: a description of the structure of the software to be implemented, the data models and structures used by the system, the interfaces between system components and the algorithms used.

Characteristics:

- Designers develop the design interactively
- Add formality and detail as they develop their design with constant backtracking to correct earlier designs
- Process activities are interleaved: feedback from one stage to another and consequent design rework
- Requires detailed knowledge about the software platform (environment in which the software will execute; e.g. operating system, database, middleware etc.)
- Activities in the design process vary, depending on the type of system being developed (see real-time systems vs. services vs. database systems)

Possible activities of the design process of information systems:

- Architectural design
    - Identify the overall structure of the system, the principal components (sub-systems, modules), their relationships and how they are distributed
- Interface design
    - Define the interfaces between system components

  - Interface specification must be unambiguous (precise interface)
  - After interface specifications are agreed, components can be designed and developed concurrently
- Component design
  - Take each system component and design how it will operate
  - May be a simple statement of the expected functionality to be implemented (specific design left to the programmer)
  - May be a list of changes to be made to a reusable component or a detailed design model
  - Design model may be used to automatically generate an implementation
- Database design
  - Design the system data structures and how these are to be represented in a database
  - Work depends on whether existing database is to be reused or a new database is to be created

Software validation

Definition: validation (verification and validation, V&V) is intended to show that a system both conforms to its specification and that is meets the expectations of the system customer

Stages in the testing process:

- Development testing
  - Components making up the system are tested by the people developing the system
  - Individual/separate tests
  - Test of functions, classes or coherent groupings of these entities
  - Test automation tools (JUnit) are commonly used (when new versions of the components are created)
- System testing
  - System components are integrated to create a complete system
  - Concerned with finding errors that result from unanticipated interactions between components and component interface problems
  - Concerned with showing that the system meets its function and non-functional requirements
  - Testing the emergent system properties
  - May be a multi-stage process for large systems
- Acceptance testing
  - Final stage in the testing process before system is accepted for operational use
  - Tested with data supplied by the customer rather than with simulated test data

Characteristics:

- Component development and testing processes are interleaved (normally)

- Incremental approach: each increment should be tested as it is developed
- Plan-driven approach: testing is driving by a set of test plans. Use of an independent team of testers
- Acceptance testing is sometimes called 'alpha testing'
- 'Beta testing' involves delivering a system to a number of potential customers who agree to use that system. They report problems to the system developers. This exposes the product to real use and detect errors that may not have been anticipated by the system builders

Software evolution

Distinction between software development and maintenance is increasingly irrelevant (software engineering as an evolutionary process where software is continually changed over tis lifetime in response to changing requirements and customer needs)

Coping with change

Reasons for change

- Business procuring the system responds to external pressures and management priorities change
- New technologies become available
- New design and implementation possibilities emerge

Rework: work that has been completed has to be redone (software development, design, requirements analysis, redesign, re-test etc.)

Approaches to reduce cost of rework:

- System prototyping
  - Version of the system or part of the system is developed quickly to check the customer's requirements and feasibility of some design decisions
  - Supports change avoidance as it allows users to experiment with the system before delivery (refine their requirements)
  - Reduced number requirements change proposals made after delivery
- Incremental delivery
  - Increments are delivered to the customer for comment and experimentation
  - Supports both change avoidance and change tolerance
  - Avoids premature commitment to requirements for the whole system and allows changes to be incorporated into later increments at relatively low costs

Refactoring: improving structure and organization of a program (an important mechanism to support change tolerance)

Prototyping

Prototype: an initial version of a software system that is used to demonstrate concepts, try out design options and find out more about the problem and its possible solutions.

Rapid, iterative development of the prototype is essential (to control costs; system stakeholders can experiment with the prototype early in the software process)

How software prototyping can help:

- Requirements engineering process: elicitation and validation of system requirements
- System design process: explore particular software solutions; support user interface design
- Get new ideas for requirements
- Find areas of strength and weakness in the software (to propose new system requirements)
- May reveal errors and omissions in the requirements
- Carry out design experiments to check the feasibility of a proposed design
- Essential part of the user interface design process (dynamic nature of user interfaces)

Delivering throwaway prototypes is usually unwise:

- May be impossible to tune the prototype to meet non-functional requirements: performance, security, robustness, reliability requirements
- Undocumented prototype due to rapid change during development; prototype code as the only design specification (bad for long-time maintenance)
- Probably degraded system structure due to changes made during prototype development. System will be difficult and expensive to maintain.
- Relaxed organizational quality standards for prototype development

Examples:

- Paper-based mock-ups of the system user interface (help users refine an interface design and work through usage scenarios)
- Wizard of Oz prototype (extension to the one above): only the user interface is developed

Incremental delivery

Definition: an approach to software development where some of the developed increments are delivered to the customer and deployed for use in an operational environment

Characteristics:

- Customers identify
    - the services to be provided by the system (in outline)
    - importance of the system's services to them

Processes

- Define outline requirements

- Assign requirements to increments
- Design system architecture
- Develop system increment
- Validate increment
- Integrate increment
- Validate system
- Deploy increment: if system is incomplete, restart with development of system increments
- Final system if system is complete

Advantages

- Customers can use early increments as prototypes and gain experience that informs their requirements for later system increments. No re-learning for users when system is available (part of the real system, unlike prototypes)
- Less waiting time for customers until entire system is delivered before they can gain value from it. Use the software immediately, which satisfies their most critical requirements
- Relatively easy to incorporate changes into the system (see incremental development)
- Most important system services receive the most testing (highest-priority services are delivered first). Customers less likely to encounter failures in the most important parts of the system

Problems with incremental delivery:

- Can be hard to identify common facilities that are needed by the increments (requirements are not defined in detail until an increment is to be implemented). Note that most systems require a set of basic facilities that are used by different parts of the system.
- Iterative development can also be difficult when a replacement system is being developed. Getting useful customer feedback is difficult (users want all the functionality of the old system and are often unwilling to experiment with an incomplete new system)
- Requires a new form of contract, which large customers (government agencies) may find difficult to accommodate
  - Specification is developed in conjunction with the software
  - No complete system specification until the final increment is specified
  - Conflicts with procurement model of many organizations: complete system specification is part of the system development contract!

Incremental development and delivery should not be used for (suffer from the same problems of uncertain and changing requirements)

- Very large systems where development may involve teams working in different locations
- Some embedded systems where software depends on hardware development
- Some critical systems where all the requirements must be analyzed to check for interactions that may compromise the safety or security of the system

Boehm's spiral model

- Is a risk-driven software process framework
- Software process is represented as a spiral
- Each loop in the spiral represents a phase of the software process
- It combines change avoidance with change tolerance
- It assumes that changes are a result of project risks and includes explicit risk management activities to reduce these risks

Each loop is split into four sectors:

- Objective setting
  - To determine specific objectives, alternatives and constraints
  - Constraints on the process and the product are identified
  - A detailed management plan is drawn up
  - Project risks are identified
  - Alternative strategies may be planned
- Risk assessment and reduction
  - A detailed analysis for each identified project risk is carried out
  - Measurements to reduce these risks
- Development and validation
  - A development model for the system is chosen
  - Throwaway prototyping: user interface risks are dominant
  - Development based on formal transformations: safety risks are the main consideration
  - Waterfall model: main identified risk is a sub-system integration
- Planning
  - Project is reviewed
  - Decision: continue with a further loop of the spiral? If so, plans are drawn up for the next phase of the project

Advantages:

- Explicit recognition of risk

Characteristics:

- Begin of cycle of the spiral: elaborating objectives such as performance and functionality
- Enumerate alternative ways of achieving these objectives and dealing with the constraints
- Development is carried out after risks have been assessed

The Rational Unified Process

- A modern process model that has been derived from work on the UML and the associated Unified software Development Process
- A hybrid process model

- Brings together elements from all of the generic process models, illustrates good practice in specification and design and supports prototyping and incremental delivery
- A phase model that identifies four discrete phases in the software process
- Phases are more closely related to business rather than technical convers (unlike the waterfall model where phases are equated with process activities)
- Separation of phases and workflows
- Recognizes that deploying software in a user's environment is part of the process
- Phases are dynamic and have goals
- Workflows are static and are technical activities that are not associated with a single phase (used throughout the development to achieve goals of each phase)

RUP described from three perspectives:

- Dynamic perspective: shows the phases of the model over time
- Static perspective: shows the process activities that are enacted
    - Called 'workflows' in the RUP description
    - 6 core process workflows and 3 core supporting workflows
- Practice perspective: suggests good practices to be used during the process

Phases:

- Inception
    - Goal: establish a business case for the system
    - Identify all external entities (people and systems) that will interact with the system and define these interactions
    - Assess the contribution that the system makes to the business
    - Project may be cancelled after this phase if contribution is minor
- Elaboration
    - Goal: develop an understanding of the problem domain, establish an architectural framework for the system, develop the project plan and identify key project risks
    - On completion: requirements model for the system (may be a set of UML use-cases, an architectural description, and a development plan for the software)
- Construction
    - Involves system design, programming and testing
    - Parts of the system developed in parallel and integrated during this phase
    - On completion: working software system and associated documentation that is ready for delivery to users
- Transition
    - Moving the system from the development community to the user community and making it work in a real environment
    - Sometimes ignored in most software process models
    - An expensive and sometimes problematic activity

    - On completion: a documented software system that is working correctly in its operational environment

Advantage of dynamic and static views:

- Phases of the development process are not associated with specific workflows (all workflows may be active at all stages of the process)

Static workflows in the RUP:

- Business modeling
    - Business processes are modeled using business use cases
- Requirements
    - Actors who interact with the system are identified
    - Use cases are developed to model the system requirements
- Analysis and design
    - A design model is created and documented
    - Used tools: architectural models, component models, object models and sequence models
- Implementation
    - Components in the system are implemented and structured into implementation sub-systems
    - Automatic code generation from design models helps accelerate this process
- Testing
    - Carried out in conjunction with implementation (an iterative process)
    - Follows the completion of the implementation
- Deployment
    - A product release is created, distributed to users and installed in their workplace
- Configuration and change management
    - Manages changes to the system (supporting workflow)
- Project management
    - Manages the system development (supporting working)
- Environment
    - Makes appropriate software tools available to the software development team

Practice perspective and its recognized fundamental best practices (describes good software engineering practices):

- Develop software iteratively: plan increments of the system based on customer priorities
- Manage requirements: explicitly document the customer's requirements
- Use component-based architectures
- Visually model software: use UML models for static and dynamic views of the software
- Verify software quality

- Control changes to software

RUP is not suitable for

- Embedded software development

Agile software development

Fundamental characteristics:

- Processes of specification, design and implementation are interleaved
    - No detailed system specification
    - Design documentation is minimized or generated automatically by the programming environment
    - User requirements document only defines the most important characteristics of the system
- System is developed in a series of versions
    - End-users and other system stakeholders are involved in specifying and evaluating each version
- System user interfaces are often developed using an interactive development system
    - System may then generate a web-based interface for a browser or an interface for a specific platform
- Incremental development methods
- Minimize documentation by using informal communications (rather than formal meetings with written documents)

Agile methods

Typical types of system development:

- Small or medium-sized product for sale, developed by a software company
- Custom system development within an organization,
    - with a clear commitment from the customer to become involved in the development process
    - without massive external rules and regulations

Principles of agile methods:

- Customer involvement
    - Provide and prioritize new system requirements
    - Evaluate the iterations of the system
- Incremental delivery
    - Customer specifies the requirements to be included in each increment
- People not process
    - Recognize and exploit the skills of the development team

- Team members should develop their own ways of working without prescriptive processes
- Embrace change
  - Perspective: system requirements will change overtime
  - System design should accommodate these changes
- Maintain simplicity
  - Simplicity for the software and for the development process
  - Actively work to eliminate complexity from the system

Non-technical problems:

- Maintaining simplicity requires extra work
- Prioritizing changes can be extremely difficult (esp. for stakeholders; see OpenGL)
- Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods
- Many organizations (large companies) are less likely to switch to more informal working environments (investigated a lot of time to structure and define processes)
- Writing contracts between the customer and the supplier may be difficult, since software requirements documents are usually part of the contract

Questions when considering agile methods and maintenance:

- Are systems that are developed using an agile approach maintainable, given the emphasis in the development process of minimizing formal documentation?
- Can agile methods be used effectively for evolving a system in response to customer change requests?

Plan-driven and agile development

- Iteration occurs within activities with formal documents used to communicate between stages of the process
- Can support incremental development and delivery

Extreme programming

- Several new versions of a system may be developed by different programmers, integrated and tested in a day
- Requirements are expressed as scenarios (user stories), which are implemented directly as a series of tasks
- Programmers work in pairs and develop tests for each task before writing the code

Characteristics / practices:

- Incremental development by small, frequent releases of the system
- Customer involvement by continuous engagement of the customer in the development team

- Might avoid excessively long working hours by using pair programming, collective ownership of the system code and by sustainable development process
- Change is embraced through regular system releases to customers
- Maintaining simplicity by constant refactoring

Extreme programming practices:

- Incremental planning
- Small releases
- Simple design
- Test-first development
- Refactoring
- Pair programming
- Collective ownership
- Continuous integration
- Sustainable pace
- On-site customer

Testing practices

- Test-first development
    - Writing the tests before writing the code
    - Run the test as the code is being written and discover problems during development
    - Reduces interface misunderstanding
    - Task implementers have to thoroughly understand the specification to write tests for the system
    - Avoids the problem of 'test-lag'
- Incremental test development from scenarios
- User involvement in the test development and validation
    - Help develop acceptance tests for the stories that are to be implemented in the next release of the system
    - Acceptance testing is incremental
- The use of automated testing frameworks
    - Essential for test-first development
    - Tests are written as executable components before the task is implemented
    - Should be stand-alone
    - Should simulate the submission of input to be tested
    - Widely used testing framework: Junit

Pair programming

Advantages:

- Idea of collective ownership and responsibility for the system (egoless programming); team has collective responsibility for resolving these problems
- Informal review process: each line is looked at by at least two people
- Support refactoring (as a process of software improvement)
- Discuss software before development: probably have fewer false starts and less rework
- Less time spent repairing bugs by informal inspection

Agile project management

Scrum approach: a general agile method but with its focus on managing iterative development rather than specific technical approaches to agile software engineering

Phases:

- Outline planning phase
    - Establish the general objectives for the project
    - Design the software architecture
- Sprint cycles
    - Each cycle develops an increment of the system
    - Cycles: assess, select, review and develop
- Project closure phase
    - Wraps up the project
    - Completes required documentation (system help frames and user manuals)
    - Assesses the lessons learned from the project

Key characteristics of sprint cycles:

- Sprints are fixed length (2-4 weeks)
- Starting point for planning: product backlog (list of work to be done on the project)
- Assessment phase: work is reviewed, priorities and risks are assigned
    - Customer is closely involved in this process and can introduce new requirements or tasks at the beginning of each spring
- Selection phase: involves all of the project team
    - Select the features and functionality to be developed during the sprint
- Develop the software
    - Short daily meetings involving all team members to review progress and reprioritize work (if required)
    - Team is isolated from the customer and the organization
    - Communications are channeled through the 'Scrum master' (to protect the development team from external distractions)
    - No specific suggestions on how to write requirements, test-first development etc.
- End of sprint: work is reviewed and presented to stakeholders

Idea behind Scrum:

- Empower the whole team to make decisions
- Deliberately avoid the term 'project manager'
- 'Scrum master': a facilitator who arranges daily meetings, tracks the backlog of work to be done, records decisions, measures progress against the backlog and communicates with customers and management outside of the team

Advantages:

- Product is broken down into a set of manageable und understandable chunks
- Unstable requirements do not hold up progress
- Whole team has visibility of everything (improves team communication)
- Customers see on-time delivery of increments (gain feedback on how the product works)
- Established trust between customers and developers

Scaling agile methods

Characteristics of large software system development:

- Collections of separate, communicating systems (in different places and time zones)
- 'brownfield systems': include and interact with a number of existing systems (political issues can be significant)
- Significant fraction of the development is concerned with system configuration rather than original code development (not necessarily compatible with incremental development and frequent system integration)
- Constrained by external rules and regulations limiting the way that they can be developed (system document required somehow)
- Long procurement and development time
- Diverse set of stakeholders (practically impossible to involve all of these different stakeholders in the development process)

Two perspectives on the scaling of agile methods:

- 'scaling up'
    - Concerned with using these methods for developing large software systems that cannot be developed by a small team
- 'scaling down'
    - Concerned with how agile methods can be introduced across a large organization with many years of software development experience

Critical adaptions:

- Up-front design and system documentation:
    - software architecture has to be designed
    - documentation to describe critical aspects of the system (database schemas, work breakdown across teams)

- cross-team communication mechanisms
    - regular phone and video conferences
    - short electronic meetings
    - communication channels: emails, instant messaging, wikis, social networking systems
- continuous integration
    - practically impossible: whole system is build every time any developer checks in a change
    - Essential: maintain frequent system builds and regular releases of the system
    - New configuration management tools that support multi-team software development have to be introduced

Difficulties in introducing agile methods to large companies:

- Project managers: reluctant to accept the risk of a new approach (if they do not have experience of agile methods, and since they do not know how this will affect their particular projects)
- Large organizations: have quality procedures and standards for all of their projects
- A wide range of skills and abilities: agile methods seem to work best when team members have a relatively high skill level
- Cultural resistance to agile methods (organizations with long history of using conventional system engineering processes)

Requirements engineering

Requirements: descriptions of what the system should do> the services that it provides and the constraints on its operation.

Requirements Engineering (RE): the process of finding out, analyzing, documenting and checking these services and constraints

User requirements: statements of what services the system is expected to provide to system users and the constraints under which it must operate, in a natural language plus diagrams

System requirements: more detailed descriptions of the software system's functions, services and operational constraints. The system requirements document / functional specification defines exactly what is to be implemented. It may be part of a contract between the system buyer and the software developers.

Functional and non-functional requirements

- Functional requirements:
    - Statements of services the system should provide
    - How the system should react to particular inputs
    - How the system should behave in particular situations
    - Optional: what the system should do
    - Examples: security (user authentication facilities in the system)
- Non-functional requirements:

- Constraints on the services or functions offered by the system
- Include constraints concerning timing, development process and standards
- Often apply to the system as a whole (unlike individual features or services)
- Example: security (limiting access to authorized users)

Functional requirements

- Vary from general to very specific requirements (what the system should do vs. local ways of working/organization of an existing system)
- May by expressed as user or system requirements or both
- Imprecisions may cause software engineering problems (see ambiguous user requirements)
- Function requirements specification should be complete and consistent (no contradictory definitions) (
- Completeness and consistency practically impossible for large complex systems
    - Easy to make mistakes and omissions when writing specifications for complex systems
    - Many stakeholders in a large system (they often have inconsistent needs)

Non-functional requirements

- May relate to emergent system properties: reliability, response time and store occupancy
- May define constraints on the system implementation: I/O device capabilities, data representations used in interfaces
- Specify or constrain characteristics of the system as a whole: performance security, availability
- Often more critical than individual function requirements
- Whole system may be unusable when failing to meet a non-functional requirement
- Often more difficult to relate components to nonfunctional requirements
- Implementation of these requirements may be diffused
    - May affect the overall architecture of a system (rather than the individual components)
    - May generate a number of related functional requirements that define new system services that are required (e.g. security)
    - May also generate requirements that restrict existing requirements
- Arise through user needs: budget constraints, organizational policies, need for interoperability with other software/hardware, external factors (safety regulations, privacy legislation)
- May come from
    - Product requirements: specify/constrain the behavior of the software
        - Usability requirements
        - Efficiency requirements (performance, space)
        - Dependability requirements
        - Security requirements
    - Organizational requirements: broad system req. derived from policies and procedures in the customer's and developer's organization.
        - Environmental requirement (operating environment of the system)

        - Operational process requirement (how the system will be used)
        - Development process requirement (programming language, development environment, process standards)
    - External requirements: from factor external to the system and its development process
        - Regulatory requirements (what must be done for the system to be used by a regulator)
        - Ethical requirements (ensure the system will be acceptable to its user and the general public)
        - Legislative requirements (ensure the system operates within the law)

Metrics for specifying non-functional requirements: speed, size, ease of use, reliability, robustness, portability

The software requirements document

Synonyms: software requirements specification, SRS

Definition: official statement of what the system developers should implement. It should include user and system requirements.

Diverse set of users:

- System customers who specify the req. and read them to check that they meet their needs (they also may make changes)
- (Senior) management of the organization that is paying for the system
- System Engineers responsible for developing the software
- System test engineers to develop validation tests for the system
- System maintenance engineers to understand the system and the relationships between its parts

Structure of a requirements document:

- Preface
    - Define the expected readership of the document
    - Describe its version history
    - A rational for the creation of a new version
    - A summary of the changes made in each version
- Introduction
    - Describe the need for the system
    - Describe the system's functions
    - Explain how it will work with other systems
    - Describe how the system fits into the overall business or strategic objectives of the organization commissioning the software
- Glossary
    - Define the technical terms used in the document

- Should not make assumptions about the experience or expertise of the reader

- User requirements definition
    - Describe the services provided for the user
    - Non-functional system requirements should also be described
    - May use natural language, diagrams or other notations that are understandable to customers
    - Product and process standards should be specified (if applicable)
- System architecture
    - A high-level overview of the anticipated system architecture
    - Showing the distribution of functions across system modules
    - Architectural components should be highlighted (that are reused)
- System requirements specification
    - Describe the functional and non-functional requirements in more detail
    - Optional: further detail to the non-functional requirements
    - Optional: interfaces to other systems
- System models
    - Include graphical system models showing the relationships between the system components, the system and its environment
    - Examples: object models, data-flow models, semantic data models
- System evolution
    - Describe the fundamental assumptions on which the system is based
    - Any anticipated changes due to hardware evolution, changing user needs etc.
    - Useful for system designers: may help to avoid design decisions
- Appendices
    - Provide detailed, specific information that is related to the application being developed
    - Example: hardware and database descriptions
    - Hardware requirements: minimal and optional configurations for the system
    - Database requirements: logical organization of the data used by the system and the relationships between data
- Index
    - A normal alphabetic index
    - An index of diagrams
    - An index of functions etc.

Requirements specification

- Should be clear, unambiguous, easy to understand, complete, consistent
- Use of natural language, simple tables, forms and intuitive diagrams
- A complete and detailed specification of the whole system

How to write a system requirement specification:

- Natural language

- Structured natural language
- Design description languages
- Graphical notations
- Mathematical specifications

Natural language specification

Guidelines:

- Is expressive, intuitive and universal
- Use a standard format
- Use language consistently to distinguish between mandatory and desirable requirements
- Use text highlighting
- Do not assume that readers understand technical software engineering language
- Try to associate a rationale with each user requirement

Structures specifications

- A way of writing system requirements where the freedom of the requirements writer is limited and all requirements are written in a standard way
- Maintains most of the expressiveness and understandability of natural language
- Ensures that some uniformity is imposed on the specification
- Define one or more standard templates for requirements and represent these templates as structured forms

Information in a standard form for specifying functional requirements:

- Description of the function or entity being specified
- Inputs and where these come from
- Outputs and where these go to
- Information about the information that is needed for the computation or other entities in the system that are used
- Action to be taken
- Pre- and post-condition before and after the function is called
- Side effects of the operation

Requirements engineering processes

High-level activities:

- Feasibility study
- Elicitation and analysis
- Specification
- Validation

Characteristics of a spiral view

- Activities are organized as an iterative process around a spiral
- Output is the system requirements document
- Amount of time and effort for each activity depends on current stage and type of system to be developed

Requirements engineering: the process of applying a structured analysis method (e.g. object-oriented analysis).

- Involves analyzing the system and developing a set of graphical system models (case models for system specification)
- Set of models describes the behavior of the system (annotated with additional information; e.g. system's required performance or reliability)

Requirements elicitation: a human-centered activity (unlike constraints by rigid system models)

Requirements management: process of managing changing requirements, e.g. modifications to the system's hardware, software or organizational environment.

Requirements elicitation and analysis

- Software engineers work with customers and system end-users to find out about
    - the application domain
    - what services the system should the system provide
    - the required performance of the system
    - hardware constraints

The requirements elicitation and analysis process:

- Requirements discovery
    - Interact with stakeholders of the system to discover their requirements (domain requirements and documentation)
    - Several complementary techniques available
- Requirements classification and organization
    - Takes unstructured collection of requirements
    - Groups related requirements
    - Organizes them into coherent clusters
    - Most common way: model of the system architecture to identify sub-systems and to associate requirements with ach sub-system
    - Practice: requirements engineering and architectural design hard to separate
- Requirements prioritization and negotiation
    - Conflict of requirements with several stakeholders
    - Prioritize requirements
    - Find and resolve requirements conflicts through negotiation

   - Compromises inevitable
- Requirements specification
   - Document requirements (formal or informal)

Eliciting and understanding requirements from stakeholders is difficult:

- Difficult to articulate
- Possibly unrealistic demands (stakeholders may don't know what is and isn't feasible)
- Stakeholders might not know what they want from a computer system (expect in most general terms)
- Requirements engineers may not understand the naturally expressed requirements of stakeholders (in their own terms and with implicit knowledge)
- Different stakeholders have different requirements
- Political factors
- Dynamic economic and business environment (where the analysis takes place)

Requirements discovery

- Synonym: requirements elicitation
- The process of gathering information about the required system and existing systems
- Distilling the user and system requirements from this information
- Sources of information: documentation, system stakeholders, specifications of similar systems (plus interviews, observations)

Interviewing

- Formal and informal interviews
- Closed interviews: stakeholder answers a pre-defined set of questions
- Open interviews
   - No pre-defined agenda
   - Requirements engineering team explores a range of issues with system stakeholders
   - Develop a better understanding of their needs
- Practice: mixture of open and closed interviews
- Part of most requirements engineering processes
- Questions to stakeholders about the system that they currently use and the system to be developed
- Difficult to elicit domain knowledge through interviews (possibly)
   - Impossible to discuss domain requirements without using terminology. Inadequate knowledge leads to misunderstandings
   - 'basis'/fundamental (but professional) knowledge might be difficult to explain to non-professionals (in terms of software engineering or computer science)
- Ineffective technique for eliciting knowledge about organizational requirements and constraints
- Characteristics of effective interviews

  - Open-minded: avoid pre-conceived ideas about the requirements (surprising requirements)
  - Prompt the interviewee to get discussions (using a springboard questions, requirements proposal, b working together on a prototype system)
- Should be used with other requirements elicitation techniques (not to miss essential information)

Scenarios

- Often easier to relate to real-life examples rather than abstract descriptions
- Useful for adding detail to an outline requirements description
- Covers one or a small number of possible interactions
- Starts with an outline of the interaction
- May include
  - What the system and users expect when the scenario start
  - Normal flow of events in the scenario
  - What can go wrong and how this is handled
  - Information about other activities going on at the same time
  - System state when the scenario finishes
- Involves working with stakeholders to identify scenarios and to capture details to be included in these scenarios
- May be written as texts, supplemented by diagrams, screen shots etc.

Use cases

- A requirements discovery technique that were first introduced in the Object method
- A fundamental feature of the unified modeling language
- Identifies the actors involved in an interaction and names the type of interaction (supplemented by additional information describing the interaction with the system)
- Documented using a high-level use case diagram
- No hard and fast distinction between scenarios and use cases
- Identify the individual interactions between the system and its users or other systems
- Should be documented with a textual description (may be linked to other models in the UML)
- Effective technique (along with scenarios) for eliciting requirements form stakeholders who interact directly with the system
- UML is a de facto standard for object-oriented modeling

Ethnography

- Software systems are used in social and organizational contexts (may constrain or influence software system requirements)
- Successful systems: satisfy social and organizational requirements

- Definition: an observational technique that can be used to understand operational processes and help derive support requirements for these processes
- Effective for discovering 2 types of requirements:
    - Requirements that are derived from the way in which people actually work (rather than the way in which process definitions say they ought to work)
    - Requirements that are derived from cooperation and awareness of other people's activities
- Can be combined with prototyping
- Ethnographic studies can reveal critical process details that are often missed by other requirements elicitation techniques)

Requirements validation

- The process of checking that requirements actually define the system that the customer really wants
- Overlaps with analysis: it is concerned with finding problems with the requirements
- Important because errors in a requirements document can lead to extensive rework costs
- Process should not be underestimated

Checks during the validation process:

- Validity checks
- Consistency checks
    - Requirements in the document should not conflict
- Completeness checks
    - Requirements that define all functions and constraints intended by the system user
- Realism checks
    - Check to ensure that requirements can actually be implemented
- Verifiability
    - System requirements should always be written so that they are verifiable
    - Reduces the potential for dispute between customer and contract

Requirements techniques to be used individually or in conjunction:

- Requirements reviews
    - Systematic analysis by a team of requires to check for errors and inconsistencies
- Prototyping
    - Demonstration of an executable model of the system
    - Personal experimentation to check if it meets their real needs
- Test-case generation
    - Requirements should be testable
    - Difficult or impossible design often means difficult implementation

Requirements management

Definition: the process of understanding and controlling changes to system requirements

- Requirements for large systems are always changing. Reasons:
    - These systems are usually developed to address 'wicked' problems (software requirements are bound to be incomplete)
- Keep track of individual requirements
- Maintain links between dependent requirements
- Establish a formal process for making change proposals and linking these to system requirements
- Start planning how to manage changing requirements during the requirements elicitation process

Reasons why change is inevitable:

- Business and technical environment of the system always changes after installation
- People how pay for a system and the users of that system are rarely the same people
- Large systems usually have a diverse user community
    - Many users have different requirements and priorities that may be conflicting or contradictory
    - Final system requirements: a compromise

Requirements management planning

- Establishes the level of requirements management detail that is required
- Required decisions to be made
    - Requirements identification: each requirements must be uniquely identified
    - A change management process: set of activities that assess the impact and cost of changes
    - Traceability policies: define the relationships between each requirement and between the requirements and the system design that should be recorded
    - Tool support: process large amounts of information about the requirements (specialist requirements management systems, spreadsheets, simple database systems)
- Required tool support for
    - Requirements storage: requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process
    - Change management: process of change management is simplified if active tool support is available
    - Traceability management: some tools help discover possible relationships between requirements

Requirements change management

- Should be applied to all proposed changes to a system's requirements (after they have been approved)
- Essential because
  - Need to decide if the benefits of implementing new requirements are justified by the costs of implementation
- Advantage of a formal change management
  - Change proposals are treated consistently
  - Changes to the requirements document are made in a controlled way

Principal stages to a change management process:

- Problem analysis and change specification
- Change analysis and costing
- Change implementation

System modeling

Definition: the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.

- Represent the system through graphical notation (UML) or formal models (mathematics)
- Used during requirements engineering process: help derive the requirements for a system
- Used during design process: describe the system to engineers implementing the system and after implementation to document the systems structure and operation

System models:

- Abstraction of the system; leaves out details

Perspectives for creating models:

- External perspective: model the context/environment of the system
- Interaction perspective: model the interactions between a system and its environment or between the components of a system
- Structural perspective: model the organization of a system or the structure of the data
- Behavioral perspective: model the dynamic behavior of the system and how it responds to events

Type of diagrams representing the essentials of a system:

- Activity diagrams: activates involved in a process or in data processing
- Case diagrams: interactions between a system and its environment
- Sequence diagrams: interactions between actors and the system and between system components
- Class diagrams: object classes in the system and the associations between these classes

- State diagrams: how the system reactions to internal and external events

Why graphical models are used:

- A means of facilitating discussions about an existing or proposed system
- A way of documenting an existing system
- A detailed system description that can be used to generate a system implementation

Context models

- Normally show that the environment includes several other automated systems
- Do not show types of relationships between the systems in the environment and the system to be specified
- Used along with other models (business process models)

UML activity diagrams:

- Show the activities that make up a system process
- Show the flow of control from one activity to another
- Start of process: indicated by a filled circle
- End of process: filled circle inside another circle
- Activities: rectangles with round corners
- Arrows represent the flow of work from one activity to another
- Arrows annotated with guards: indicate the condition when that flow is taken
- Solid bar: indicate activity coordination

Interaction models

Some types of interaction:

- User interaction: user inputs and outputs
- System interaction: systems and components

Modeling user interaction:

- Helps to understand if a proposed system structure is likely to deliver the required system performance and dependability

Use case modeling

- Used to model interactions between a system and external actors (users, other systems)
- Used to support requirements elicitation
- Gives a fairly simple overview of an interaction (you have to provide more detail to understand what is involved: textual description, structured description in a table, sequence diagram)

Use case:

- Represents a discrete task (involves external interaction with a system)
- Shown as an ellipse with the actors involved in the use case

Key elements often involved in use cases: actors, description, data, stimulus, response, comments

Sequence diagrams

- Used to model the interactions between
  - the actors and the objects in a system
  - the objects themselves
- Shows the sequence of interactions that take place during a particular use case or use case instance
- Objects and actors involved are listed along the top of the diagram
- Interactions between objects are indicated by annotated arrows
- Rectangle on the dotted lines indicates the lifeline of the object concerned
- Read the sequence of interactions from top to bottom
- Annotations on the arrows indicate the calls to the objects, their parameters and the return values
- Include every interaction in sequence diagrams for code generation or detailed documentation

Structural models

- Display the organization of a system in terms of the components that make up that system and their relationships
- Static models: show the structure of the system design
- Dynamic models: show the organization of the system when it is executing
  - May be very different from a static model of the system components (for a dynamic organization of a system as a set of interacting threads)
- Creation of structural models:
  - Through discussions and system architecture designs

Class diagrams

- Used when developing an object-oriented system model to show the classes in a system and the associations between these classes
- Association: a link between classes that indicates that there is a relationship between these classes
  - Each class may have some knowledge of its associated class
- Expressed at different levels of detail
  - World: develop of model (essential objects)
- Types of relationship: 1 to 1, m to 1, m to n

Generalization

- An everyday technique that we use to manage complexity

- Easier to identify common characteristics
- Possible to make general statements that apply to all class members
- UML has a specific type of association to denote generalization
    - Shown as an arrowhead pointing up to the more general class
    - Attributes and operations associated with higher-level classes are also associated with the lower-level classes
    - Lower-level classes are subclasses inherit the attributes and operations from their superclasses
    - Lower-level classes add more specific attributes and operations

Aggregation

- UML provides a special type of association between classes called aggregation
- One object is composed of other objects
- Use a diamond shape next to the class that represents the whole

Behavioral models

- Show what happens or what is supposed to happen when a system responds to a stimulus from its environment
    - Data: data arrives that has to be processed by the system
    - Events: event happens that triggers system processing (may have associated data)
- Dynamic behavior of the system (as it is executing)

Data-driven modeling

- Show the sequence of actions involved in processing input data and generating an associated output
- Useful during the analysis of requirements as they can be used to show end-to-end processing in a system
- Data-flow models:
    - Illustrate processing steps in a system
    - Tracking and documenting how the data associated with a particular process moves through the system
    - Helps analysts and designers understand what is going on
    - Simple and intuitive
    - Possible to explain them to potential system users who can then participate in validating the model
    - Focus on system functions
    - Do not recognize system objects (therefore, data flow models not included in UML)
    - Similar to activity diagrams in UML 2.0 (data-driven systems are common in business)
    - Highlight functions
- Alternate way: use UML sequence diagrams
    - Highlight objects in a system

Event-driven modeling

- Shows how a system responds to external and internal events
- Based on the assumption that
    - A system has a finite number of states
    - Events may cause a transition from one state to another
- Particularly appropriate for real-time systems
- State diagrams (UML):
    - Show system states and events that cause transitions from one state to another
    - Do not show the flow of data within the system
    - May include additional information on the computations carried out in each state
    - Rounded rectangles: represent system states (may include a brief description of the actions taken in that state)
    - Labeled arrows: represent stimuli that force a transition
    - Optional: start and end states using filled circles (see activity diagrams)

Model-driven engineering

- An approach to software development where models rather than programs are the principal outputs of the development process
- Programs (execute on hardware/software platform) are generated automatically from models
- Raises level of abstraction in software engineering (no longer have to concerned with programming language details or specifics of execution platforms)
- Roots in model-driven architecture (MDA); was proposed by the Object Management Group (OMG)
- Concerned with all aspects of the software engineering process (including model-based requirements engineering, software processes for model-based development, model-based testing)

Advantages:

- Allows engineers to think about systems at a high level of abstraction
- No concern for the details of their implementation
- Reduces the likelihood of errors
- Speeds up the design and implementation process
- Allows for the creation of reusable, platform-independent application models
- Only a translator for a platform is required to adapt the system to some new platform technology

Disadvantages:

- Abstractions that are supported by the models are not always the right abstractions for implementations (using an off-the-shelf, configurable package)

- Arguments for platform independence are only valid for large long-lifetime systems where the platforms become obsolete during a system's lifetime (for this class of systems, implementation is not the major problem)
- Requirements engineering, security and dependability, integration with legacy systems, and testing are more significant
- Not clear whether the costs and risks of model-driven approaches outweigh the possible benefits

Model-driven architecture

- Is a model-focused approach to software design and implementation
- Uses a sub-set of UML models to describe the system
- Models at different levels of abstraction are created
- Generate a working program without manual intervention from a high-level platform independent model (in principle)
- Complete automated translation tools are unlikely to be available in the near future
- Translation of PIMs to PSMs is more mature (several commercial tools exists)
- An uneasy relationship between agile methods and model-driven architecture

Types of abstract system models

- Computation independent model (CIM):
  - Models the important domain abstractions used in the system
  - Sometimes called domain models
- Platform independent model (PIM):
  - Models the operation of the system without reference to its implementation
  - Usually described using UML models that show the static system structure and how it responds to external and internal events
- Platform specific models (PSM):
  - Transformations of the platform-independent model with a separate PSM for each application platform
  - Layers of PSM may add some platform-specific detail
  - First-level PSM: middleware-specific, database independent

Executable UML

- Have to be able to construct graphical models whose semantics are well defined
- Need to add information to graphical models about the ways in which the operations defined in the model are implemented
- Supported by xUML or Executable UML
- UML is more concerned with software design and expressiveness, rather than programming language or semantic details of the language

- Number of model types has been dramatically reduced to 3 key model types (to create executable sub-set of UML):
  - Domain models: identify the principal concerns in the system (UML class diagrams with objects, attributes and associations)
  - Class models: classes are defined (along with their attributes and operations)
  - State models: a state diagram is associated with each class and is used to describe the lifecycle of the class
- Dynamic behavior of the system may be
  - Specified declaratively using the object constraint language (OCL)
  - Expressed using UMLs action language (a very high-level programming language)

Architectural design

- Concerned with understanding how a system should be organized and designing the overall structure of that system
- The first stage in the software design process
- The critical link between design and requirements engineering (identifies the main structural components in a system and the relationships between them)
- Output: architectural model (describes how the system is organized as a set of communicating components)
- A significant overlap between the processes of requirements engineering and architectural design (in practice) (a system specification should not include any design information)
- Different levels of abstraction
  - Individual program architectures
  - Distributed system architectures (composed of several programs, components)
- Affects performance, robustness, disreputability, maintainability of a system
- Modeled using
  - Simple block diagrams (a box represents a component)
  - Arrows (signals or data flow between components)

Advantages of explicitly designing and documenting software architectures:

- Stakeholder communication (high-level presentation; focus discussions)
- System analysis (effect on design’s critical requirements: performance, reliability, maintainability)
- Large-scale reuse (system architecture often the same for systems with similar requirements)

Ways an architectural model is used:

- Facilitate discussions about the system design
- Document an designed architecture

Architectural design decisions

- A creative process: design a system organization that will satisfy the functional and non-functional requirements of a system
- Activities within the process depend on the
  - Type of system being developed
  - Background and experience of the system architect
  - Specific requirements for the system
- A series of decisions to be made rather than a sequence of activities
- Architecture may be based on a particular pattern or style

Architectural pattern

- A description of a system organization (client-server organization, layered organization)
- Capture the essence of an architecture that has been used in different software systems

Architectural style

- Choose most appropriate structure (client-server, layered)

Architectural pattern and style should depend on the non-functional system requirements:

- Performance
  - Localize critical operations within a small number of components
  - Deployed on the same computer (rather than distributed)
  - Use a few relatively large components and fine-grain components
  - Reduce component communications
  - Consider run-time system organizations (allows the system to be replicated and executed on different processors)
- Security
  - Layered structure should be used
  - Most critical assets protected in the innermost layers
  - A high level of security validation applied to these layers
- Safety
  - Safety-related operations should be located in either a simple components or in a small number of components
  - Reduces the cost and problems of safety validation
  - Provides related protection systems (safely shut down system if failure)
- Availability
  - Include redundant components (replace and update components without stopping the system)
- Maintainability
  - Use fine-grain, self-contained components
  - Separate producers of data from consumers

  - Shared data structures should be avoided

Architectural views

- Impossible to represent all relevant information about a system's architecture in a single architectural model
- Views might be:
  - How the system is decomposed into modules
  - How the run-time processes interact
- Logical view
  - Key abstractions in the system (as objects, object classes)
  - Relate system requirements to entities in this logical view
- Process view
  - How system is composed of interacting processes (at run-time)
  - Useful for making judgments about non-functional system characteristics (performance, availability)
- Development view
  - How the software is decomposed for development
  - Shows the breakdown of the software into components (implemented by a single developer or developer team)
  - Useful for software managers and programmers
- Physical view
  - How software components are distributed across the processors in the system
  - Shows system hardware
  - Useful for system engineers planning a system deployment
- Conceptual view
  - Abstract view of the system
  - Bases for decomposing high-level requirements into more detailed specifications
  - Help engineers make decisions about components (that can be reused)
  - Represent a product line rather than a single system
  - Used to support architectural decision making
  - A way of communicating the essence of a system to different stakeholders
  - Possible to associate architectural patterns with different view of a system
- Languages to describe views
  - UML (might remain most commonly used way of documenting system architectures)
  - ADLs (specialized architectural description languages) (components and connectors, rules, guidelines)

Architectural patterns

- Patterns as a way of presenting, sharing and reusing knowledge about software system is widely used
- Patterns for organizational design, usability, interaction, configuration management

- A stylized, abstract description of good practice (tried and testing in different systems and environments)
- Describe a system organization that has been successful in previous systems
- Include information of when it is and is not appropriate to use that pattern, the pattern's strengths and weaknesses

Model-View-Controller (MVC):

- Basis of interaction management in many web-based systems

Layered architecture

- Separation and independence allow changes to be localized
- System functionality: organized into separate layers
- Each layer relies on the facilities and services offered by the layer immediately beneath it
- Supports incremental development of systems
- Architecture: changeable and portable (layers can be easily replaced as long as the interface is constant)
- Easier to provide multi-platform implementations of an application system (machine dependencies in inner layers)
- Disadvantages:
    - Clean separation between layers often difficult
    - High-level layer may have to interact directly with lower-level layers (rather than through the layers immediately below)
    - Performance penalty (multiple levels of interpretation of a service request)
- Sample: a generic layered architecture
    - User interface
    - User interface management; authentication and authorization
    - Core business logic; application functionality; system utilities
    - System support (OS, database etc.)
- Sample: LIBSYS
    - 5 layers
    - Allows controlled electronic access to copyright material from a group of university libraries

Repository architecture

- All data in a system is managed in a central repository (accessible to all system components)
- Components do not interact directly (through repository instead)
- When to use
    - System with large volumes of information generated (stored for a long time)
    - Data-driven systems (inclusion of data in the repository triggers an action or tool)
- Advantages

    - Components can be independent
    - Changes made by one component can be propagated to all components
    - All data can be managed consistently (in one place)
- Disadvantages
    - Single point of failure: problems in the repository affect the whole system
    - Inefficiencies in organizing all communication through the repository
    - Difficult to distribute the repository across several computers
- Sample: architecture for an IDE
    - UML editors
    - Code generators
    - Java editor
    - Python editor
    - Report generator
    - Design analyzer
    - Design translator

Client-server architecture

- System's functionality is organized into services (each services delivered from a separate server)
- Clients: users of these services; they access servers to make use of them
- When to use
    - Data in a shared database has to be accessed from a range of locations
    - Load on a system is variable (servers can be replicated)
- Advantages
    - Servers can be distributed across a network
    - General functionality can be available to all clients (does not need to implemented by all services)
- Disadvantages
    - Each service is a single point of failure
    - Susceptible to denial of service attacks or server failure
    - Performance: unpredictable (depends on the network and system)
    - Management problems possible (servers owned by a different organization)
- Sample: video/DVD library
    - Catalogue server
    - Video server
    - Picture server
    - Web server

Pipe and filter architecture

- Run-time organization of a system
- Functional transformations process their inputs and produce outputs
- Data flows from one to another and is transformed as it moves through the sequence

- Each processing component (filter) is discrete (carries out one type of data transformation)
- Wen to use
    - Data processing applications (batch- and transaction-based)
    - Applications where inputs are processed in separate stages to generate related outputs
- Advantages
    - Easy to understand
    - Supports transformation reuse
    - Workflow style matches structure of many business processes
    - Evolution by adding transformation is straightforward
    - Implemented as sequential or concurrent system
- Disadvantages
    - Format of data transfer has to be agreed upon between communicating transformations
    - Each transformation must parse input and unparsed output to the agreed form
    - Increases system overhead
    - Possibly impossible to reuse functional transformations that use incompatible data structures

Application architectures

- Intended to meet a business or organizational need
- Business characteristics: Hire people, issue invoices, keep accounts etc.
- Business operating: use common sector-specific applications
- Encapsulate the principal characteristics of a class of systems
- May be re-implemented when developing new systems
- Possible application reuse without reimplementation for many business systems (ERP, SAP, Oracle, COTS)
- Identify stable structural features of the system architectures

Ways of using models of application architectures:

- As a starting point for the architectural design process
    - If unfamiliar with type of application
    - Base initial design on a generic application architecture (specialization required)
- As a design checklist
    - Compare developed architectural design with generic application architecture
    - Check consistency
- As a way of organizing the work of the development team
    - Possible to develop application architectures in parallel
    - Assign work to group members to implement components within the architecture
- As a means of assessing components for reuse
    - Compare available components with generic structures
    - Comparable components in the application architecture?
- As a vocabulary for talking about types of applications

  - Use concepts identified in the generic architecture to talk about the applications

Transaction processing systems

- Abbreviation: TP systems
- Designed to process user requests for information from a database, or requests to updated a database
- Database transaction: a sequence of operations that is treated as a single unit (an atomic unit) (and all of the operations have to be completed before the database changes are made permanent)
- Prevent inconsistencies in the database
- Transaction: a coherent sequence of operations that satisfies a goal
- TP systems are usually interactive systems (users make asynchronous requests for services)
- Possible organization: pipe and filter architecture
- Sample: ATM system

Information systems

- Allows controlled access to a large base of information
- Samples: library catalog, flight timetable, records of patients in a hospital
- Increasingly web-based systems
- Possible implementation: layered architecture
  - Top layer: user interface (UI)
  - Second layer: UI functionality (delivered through web browser) (log in, checking components, data validation components, menu management components)
  - Third layer: functionality of the system (components of system security etc.)
  - Lowest layer: database management system
- Often implemented as multi-tier client-server architectures
  - Web server: responsible for all user communications
  - Application server: application-specific logic, information storage, information retrieval request
  - Database server: moves information to/from database, handles transaction management

Language processing systems

- Translate natural/artificial languages into another representation of that language
- May execute resulting code for programming languages (e.g. script languages)
- Samples:
  - Compilers translate a programming language into machine code
  - Translate an XML data description into commands to query a database to an alternative XML representation
  - Natural language processing systems may translate one natural language to another

- Possible architecture (programming language)
    - Source language instructions: define the program to be executed
    - Translator: converts source language instructions into instructions for an abstract machine
    - Interpreter: fetches the instructions for execution; executes them using data from the environment (if required) (usually hardware unit processing machine instructions or software component)
    - Output of the process: result of interpreting the instructions on the input data

Pipe and filter compiler architecture (components):

- Lexical analyzer
    - Takes input language tokens
    - Converts them to an internal form
- Symbol table
    - Holds information about the names of entities (variables, class names, object names etc.)
- Syntax analyzer
    - Checks syntax of the language being translated
    - Uses a defined grammar of the language
    - Builds a syntax tree
- Syntax tree
    - An internal structure representing the program being compiled
- Semantic analyzer
    - Uses information from the syntax tree and symbol table
    - Checks semantic correctness of the input language text
- Code generator
    - Walks the syntax tree
    - Generates abstract machine code
- Additional components:
    - Analyze and transform the syntax tree (improve efficiency and remove redundancy from generated machine code)
    - Dictionaries

Design and implementation

- Develop and executable software system
- A stage in software engineering process
- Software design and implementation activities are invariably interleaved
- Software design
    - Creative activity
    - Identify software components and their relationships
    - Take implementation issues into account when developing a design

- Software implementation
    - Process of realizing the design as a program
- Important implementation decision
    - Buy or build the application software? (COTS vs. open-source)

Object-oriented design using the UML

- Object-oriented system
    - Made up of interacting objects that maintain their own local state (provide operations on that state)
- Design object classes and the relationships between them
- What to do to develop a system design
    - Understand and define the context and external interactions with the system
    - Design the system architecture
    - Identify the principal objects in the system
    - Develop design models
    - Specify interfaces
- Design is not a clear-cut, sequential process (ideas, propose solutions, refine solutions, ideas etc.)

System context and interactions

- Understand relationships between software and external environment
- Decide how to provide required system functionality, how to structure the system to communicate with its environment
- Set the system boundaries helps to decide what features are implemented
- System context: a structural model that demonstrates other systems in the environment
    - May be represented using associations (relationships)
    - Nature of relationships are specified
    - Using a simple block diagram
- Interaction model: a dynamic model that shows how the system interacts with its environment
    - Model with an abstract approach (not too much detail)
    - Use a case model
    - Cases should be described in structured natural language

Architectural design

- Base knowledge: system context and interactions plus general knowledge of principles of architectural design (with more detailed domain knowledge)
- Identify major components and their interactions
- Organize components using an architectural pattern (layered, client-server etc.; optional)

Object class identification

- Prerequisites:

    - Ideas about the essential objects in the system to be designed
- High-level system objects are needed to encapsulate the system interactions defined in the use cases (usually)
- Methods how to identify object classes
    - Grammatical analysis of a natural language description of the system to be constructed (nouns: objects, attributes; verbs: operations, services)
    - Tangible entities in the application domain (aircraft, roles, events, interactions, locations etc.)
    - Scenario-based analysis where various scenarios of system use are identified and analyzed in turn (team responsible for the analysis must identify the required objects, attributes and operations)
- Use several knowledge sources to discover object classes
- Information from application domain knowledge or scenario analysis to refine/extend initial objects (can be collected from requirements documents, discussions with users, from analyses of existing systems)
- Focus on objects themselves (without thinking about how these might be implemented)
- Look for common features (then design inheritance hierarchy for the system)

Design models

- Show objects or object classes in the system
- Show associations and relationships between objects
- Synonym: system models
- A bridge between the system requirements and the implementation of a system
- They have to be abstract (to focus on relationships)
- They have to include enough detail for programmers (to make implementation decisions)
- Solution: develop models at different levels of detail
- Important step: decide on the needed design models and the level of detail required in these models (depends on type of system)
- With UML, develop 2 kinds of models
    - Structural models: static structure of the system using objects and relationships
    - Dynamic models: dynamic structure of the system; show interactions between system objects

Useful models for adding detail to use case and architectural models:

- Subsystem models
    - Show logical groupings of objects into coherent subsystems
    - Represented using a form of class diagram (each subsystem shown as a package with enclosed objects)
    - Subsystem models are static models (structural)
- Sequence models

  - Show sequence of object interactions
  - Represented using a UML sequence / collaboration diagram
  - Sequence models are dynamic models
- State machine models
  - Show how individual objects change their state in response to events
  - Represented using state diagrams (in the UML)
  - State machine models are dynamic models

Interface specification

- Important part of an design process
- To design objects and subsystem in parallel
- Concerned with specifying the detail of the interface to an object / group of objects
- Define signatures and semantics of the services (provided by the object / group of objects)
- Can be specified in the UML (same notation as a class diagram)
- UML stereotype <<interface>> should be included
- No attribute section
- Semantics of the interface
  - Defined using the object constraint language (OCL)
- Do not include details of the data representing in an interface design (no attributes in the specification)
- Design that is more maintainable
- Not a simple 1:1 relationship between objects and interfaces
  - One object may have several interfaces
  - Supported directly in Java
  - Group of objects may all be accessed through a single interface

Design patterns

- Pattern is
  - A description of the problem and the essence of its solution (solution may be reused in different settings)
  - Not a detailed specification
  - A description of "accumulated wisdom and experience"
  - A well-tried solution to a common problem
- Usually associated with object-oriented design
- Published patterns often rely on object characteristics (e.g. inheritance, polymorphism)
- A way of reusing the knowledge and experience of other designers
- Essential elements of design patterns
  - A name (unique, meaningful)
  - A description of the problem area (when to apply the pattern)
  - A solution description of the parts of the design solution, their relationships and their responsibilities (not a concrete design description; a template for a design solution)

  - A statement of the consequences (results, trade-offs); helps to decide whether or not to use the design pattern
- Any design problem may have an associated pattern that can be applied
- Patterns support high-level concept reuse
- A great idea, but requires experience of software design to use them effectively
  - Recognize situations where patterns can be applied
- Sample: observer pattern

Implementation issues

- Create an executable version of the software
- May involve
  - developing programs in high- or low-level programming languages
  - Tailoring and adapting generic, off-the-shelf systems to meet the specific requirements of an organization

Reuse

- Due to costs and schedule pressure
- Develop new systems more quickly, with fewer development risks and lower costs
- Software should be more reliable

Possible at a number of different levels:

- Abstraction level
  - Don't reuse software directly
  - Use knowledge of successful abstractions in the design of your software
  - Ways of representing abstract knowledge for reuse: design patterns and architectural patterns
- Object level
  - Directly reuse objects from a library (rather than writing code)
  - Find appropriate libraries and discover if objects and methods offer the functionality needed
  - Sample: process mail messages (in Java)
- Component level
  - Components: collections of objects and object classes
  - Adapt and extend components by adding some source code
  - Sample: user interface
- System level
  - Reuse entire application systems
  - Involves some kind of configuration of these systems
  - Done by adding/modifying code or by using system's own configuration interface
  - Used by most commercial systems (generic COTS systems are adapted and reused)

Costs associated with reuse:

- Costs of time spent in looking for software to reuse and assessing whether or not it meets the needs
- Costs of buying the reusable software (can be very high for large COTS systems)
- Costs of adapting and configuring the reusable software components or systems to reflect the requirements
- Costs of integrating reusable software elements with each other and with the new code that has been developed

Configuration management

- Change management is absolutely essential
- General process of managing a changing software system
- Aim: support the system integration process so that all developers can
  - Access the project code and documents in a controlled way
  - Find out what changes have been made
  - Compile and link components to create a system
- Fundamental configuration activities
  - Version management (keep track of different versions; stop overwriting code being submitted by someone else)
  - System integration (define what versions of components are used to create each version of a system; description is used to build a system automatically by compiling and linking the requirements components)
  - Problem tracking (allow users to report bugs and other problems)
- See: software configuration management tools (integrated or separate)

Host-target development

- Most software development is based on host-target model
  - Software is developed on one machine (host)
  - Software runs on separate machines (targets)
- Alternate perspective: development-execution platform
- Often use of simulators in case of development for
  - Embedded systems
  - Different platforms or expensive hard drive
- Simulators
  - Speed up development process for embedded systems
  - Are expensive to develop
  - Usually only available for most popular hardware architectures
- In case middleware is being used, the software is usually transferred to the execution platform for testing (due to license restrictions, etc.)

- Integrated tools of a software development platform (to support software engineering processes)
    - An integrated compiler and syntax-directed editing systems
    - Language debugging system
    - Graphical editing tools (edit UML models)
    - Testing tools (JUnit) to automatically run a set of tests on a new version of a program
    - Project support tools (help to organize code for different development projects)
    - Advanced tools like static analyzers
- IDE: an integrated development environment, made up of several software development tools and that support different aspects of software development
- Document decisions on hardware and software deployment using UML deployment diagrams

Issues to consider when to decide how the developed software will be deployed for distributed systems:

- Hardware and software requirements of a component
    - A component must be deployed on a platform that provides the required hardware/software support (in case the component has been designed for a specific hardware architecture or relies somehow on it)
- Availability requirements of the system
    - High-availability systems may require components to be deployed on more than one platform
    - An alternative implementation of the component is available in case of platform failure
- Component communications
    - Deploy components on the same platform (or those physically close) in case of high level of communications traffic between components
    - Reduces communications latency, delay between the time a message is sent by one component and received by another

Open source development

- An approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process
- Roots: Free Software Foundation
- Successful open source system sill rely on a core group of developers who control changes to the software
- Samples: Linux, Java, Apache, MySQL (IBM, Sun)
- Fairly cheap/free to acquire open source software
- Emerging/common business models: selling support for a product rather than selling a software product

Open source licensing

- Common licenses: GPL, LGPL, BSD

- Establish a system for maintaining information about pen source components that are downloaded and used
- Be aware of the different types of licenses and understand how a component is licensed before it is used
- Be aware of evolution pathways for components
- Educate people about open source
- Have auditing systems in place
- Participate in the open source community
- Trend:
    - Increasingly difficult to build a business by selling specialized software systems
    - Sell support and consultancy to software users for their open source software

Software testing

- Intended
    - To show that a program does what it is intended to do
    - To discover program defects before it is put into use
- Goals of the testing process
    - Validation testing: demonstrate that the software meets its requirements (to the developer and customer)
    - Defect testing: discover incorrect, undesirable or specification non-conform behavior of the software
- Usually, not all software defects or incorrect required behaviors (according to specification) are found
- Testing can only show the presence of errors, not their absence
- Part of software verification and validation (V&V) processes
    - Concerned with checking that software being developed meets its specification and delivers the functionality expected by the people paying for the software
    - May involve software inspections and reviews
- Software verification
    - Check that the software meets its stated functional and non-functional requirements
    - Establish confidence that the software is fit for purpose
- Software validation
    - A more general process
    - Ensure that the software meets the customer's expectations
    - Goes beyond simple conformance checks with the specification
    - Essential: requirements specifications do not always reflect the real wishes or needs of system customers and users
- Software purpose
    - The more critical the software, the more important that it is reliable
- User expectations

    - Usually lower for newly installed software (users may tolerate failures because the benefits of use outweigh the costs of failure recovery)
    - More thorough testing of later versions may be required as software matures
- Marketing environment
    - Competing products
    - Price that customers are willing to pay for a system
    - Required schedule for delivering that system
    - Competitive environment: release program before it has been fully tested and debugged (be the first into the market)
- Inspections and reviews (static V&V)
    - Analyze and check the system requirements, design models, the program source code and even proposed system tests
- Inspections
    - Focus on the source code of a system
    - Any readable representation can be inspected (requirements, design model)
    - Use knowledge of the system, its application domain and the programming / modeling language to discover errors
    - Cannot replace software testing (unexpected interactions, timing problems, system performance)
- Advantages of software inspection over testing
    - A single inspection session can discover many errors in a system (static process)
    - During testing, errors can mask other errors (side effects of the original error or new error)
    - Incomplete versions of a system can be inspected without additional costs
    - Can also consider broader quality attributes of a program (compliance with standards, portability, maintainability)
    - Can look for inefficiencies, inappropriate algorithms, poor programming style (could make the system difficult to maintain and update)
- Test cases
    - Specifications of the inputs to the test and the expected output from the system, plus a statement of what is being tested
- Stages of testing
    - Development testing (tested during development)
    - Release testing (separate testing team tests a complete versions before it is released)
    - User testing (potential users test the system in their own environment)
- Usually involves a mixture of manual and automated testing

Development testing

- Includes all testing activities that are carried out by the team developing the system
- Testing at 3 levels:
    - Unit testing: individual program units or object classes are tested

  - Component testing: several individual units are integrated to create composite components (should focus on testing component interfaces)
  - System testing: some or all of the components in a system are integrated and the system is tested as a whole (should focus on testing component interactions)
- Primarily a defect testing process
- Discover bugs in the software
- Usually interleaved with debugging

Unit testing

- Testing program components (methods, object classes)
- Test all operations associated with the object
- Set and check the value of all attributes associated with the object
- Put the object into all possible states (simulate all events that cause a state change)
- Automated test has three parts
  - Setup: initialize the system with the test case (inputs and expected outputs)
  - Call: call the object/method to be tested
  - Assertion: compare the results of the call with the expected result
- Use mock objects if objects to be tested depend on other, yet untested objects
  - Objects with the same interface as the external objects being used that simulate its functionality
  - Sample: a mock object simulating a database may have only a few data items that are organized in an array
  - Can be used to simulate abnormal operations or rare events

Choosing unit test cases

- Important to choose effective unit test cases (testing is expensive and time consuming)
  - Test cases should show that the component does what it is supposed to do
  - Defects in the component should be revealed by test cases
- Two kinds of test case:
  - Normal operation of the program (show that the component works)
  - Based on testing experience of where common problems arise (use abnormal inputs)
- Strategies for choosing test cases:
  - Partition testing (identify groups of inputs with common characteristics)
  - Guideline-based testing (use testing guidelines to choose test cases; guidelines reflect previous experience of errors component programmers often make)
- Equivalence partitions / domains:
  - Classes with common characteristics
  - Programs normally behave in a comparable way for all members of a class
  - Sample: positive numbers, negative numbers, menu selections
- Partition testing

  - A systematic approach to test case design
  - Identify all input and output partitions for a system / component
  - Test cases are designed so that the inputs / outputs lie within these partitions
  - Can be used for systems and components
  - Identify partitions by using the program specification or user documentation and from experience where you predict the classes of input value that are likely to detect errors
  - Black-box testing: use the specification of a system to identify equivalence partitions
  - White-box testing: look at the code of the program to find other possible tests
- Testing guidelines
  - Test software with sequences that have only a single value
  - Use different sequences of different sizes in different tests
  - Derive tests so that the first, middle and last elements of the sequence are accessed (reveals problems at partition boundaries)
- General guidelines
  - Choose inputs that force the system to generate all error messages
  - Design inputs that cause input buffers to overflow
  - Repeat the same input/series of inputs numerous times
  - Force invalid outputs to be generated
  - Force computation results to be too large / too small

Component testing

- Components are often made up of several interacting objects
- Component interface testing:
  - Parameter interfaces (data / function references are passed from one component to another)
  - Shared memory interfaces (block of memory is shared between components; often used in embedded systems)
  - Procedural interfaces (one component encapsulates a set of procedures that can be called by other components)
  - Message passing interfaces (one component requests a service from another component by passing a message to it; a return message includes the results of executing the service; see client-server systems)
- Interface errors are one of the most common forms of error in complex systems:
  - Interface misuse: a calling component makes an error in the use of the called component's interface (often found in parameter interfaces with wrong parameters)
  - Interface misunderstanding: a calling component misunderstands the specification of the interface of the called component and makes assumptions about its behavior (e.g.: binary search on an unordered array)
  - Timing errors: often occur in real-time systems using a shared memory / message-passing interface (consumer may access out-of-date information)
- General guidelines for interface testing:

- Examine the code to be tested and explicitly list each call to an external component extreme values are most likely to reveal interface inconsistencies)
- Always test the interface with null pointer parameters (where pointers are passed across an interface)
- Design tests that deliberately cause the component to fail (where a component is called through a procedural interface)
- Use stress testing in message passing systems
- Design tests that vary the order in which components that interact through shared memory are activated (may reveal implicit assumptions made by the programmer about the order in which the shared data is produced and consumed)

- Inspections and reviews can sometimes be more cost effective than testing for discovering interface errors
- Inspections can concentrate on component interfaces
- Static analyzers can detect a wide range of interface errors
- A strongly typed language (Java, C/C++) allows many interface errors to be trapped by the compiler

System testing

- Involves integrating components to create a version of the system
- Testing the integrated system
- Checks that
  - Components are compatible
  - Interact correctly
  - Transfer the right data at the right time across their interfaces
- Differences with component testing
  - Reusable components may be integrated with newly developed components (that have been separately developed and off-the-shelf systems; during system testing)
  - Components developed by different team members / groups may be integrated at this stage (a collective rather than an individual process)
- Emergent behavior when components are integrated to create a system
  - Some elements of system functionality only become obvious when put together
  - May be planned emergent behavior
  - Sometimes, its unplanned/unwanted emergent behavior (develop checks that the system is only doing what it is supposed to do)
- Should focus on testing the interactions between the components and objects that make up the system
- Optional: test reusable components / systems (check that they work as expected when they are integrated with new components)
- Should discover component bugs that are only revealed when a component is used by another
- Helps finding misunderstandings about other components (made by component developers)
- Use case-based testing is an effective approach to system testing

  - Each use case is implemented by several components / objects
  - Use case forces these interactions to occur
  - See the objects/components being involved in the interaction in a sequence diagram (to model the use case implementation)
- Exhaustive testing is impossible
- Companies may have policies for choosing subsets of possible test cases
- Automated system testing is usually more difficult than automated unit/component testing

Test-driven development

- Abbreviation: TDD
- An approach to program development in which testing and code development is interleaved
- Incremental code development alongside with test development for each increment
- Introduced as part of agile methods (Extreme Programming)
- Can also be used in plan-driven development processes
- Steps in the TDD process:
  - Identify new functionality: should normally be small and implementable in a few lines of code
  - Write test: implement this as an automated test
  - Run test: the new test will fail (have not implemented the new functionality yet)
  - Implement the functionality and re-run the test: may involve refactoring existing code to improve it and add new code to what's already there
  - Move on to implementing the next chunk of functionality (once all tests run successfully)
- An automated testing environment is essential for TDD
- Advantages:
  - Better problem understanding
  - Code coverage: all the code in the system has actually been executed by at least one associated test
  - Regression testing: run regression tests to check that changes to the program hove not introduced new bugs
  - Simplified debugging: easy to find bug/problem if a test fails (the newly written code needs to be checked and modified)
  - System documentation: tests act as a form of documentation that describe what the code should be doing
  - Reduces cost of regression testing (very expensive and often impractical when a system is manually tested)
- May be ineffective with multi-threaded systems (interleaved threads at different times in different test runs may produce different results)
- A successful approach for small and medium-sized projects
- No evidence that TDD leads to poorer quality code

Release testing

- Process of testing a particular release of a system that is intended for use outside of the development team
- Distinctions between release testing and system testing
  - A separate team should be responsible for release testing
  - System testing is defect testing; release testing is validation testing
- Convince the supplier of the system that it is good enough for use
- Usually a black-box testing process (see functional testing; tester is only concerned with functionality and not with implementation)

Requirements-based testing

- Requirements should be testable (a general principle of good requirements engineering practice)
- A systematic approach to test case design
- Consider each requirement and derive a set of tests for it
- A validation testing (rather than a defect testing)
- Does not mean just writing a single test
- Normally have to write several tests to ensure that you have coverage of the requirements
- Also maintain traceability records of your requirements-based testing (link the tests to the specific requirements that are being tested)

Scenario testing

- An approach to release testing where you devise typical scenarios of use and use these to develop test cases
- Scenario:
  - A story that describes one way in which the system might be used
  - Should be realistic
  - Real system users should be able to relate to them
  - Possible reuse scenarios as part of the requirements engineering process
- Should be a narrative story that is credible and fairly complex
- Should motivate stakeholders (they should believe that it is important that the system passes the test)
- Should be easy to evaluate
- Normally testing several requirements within the same scenario (automatically checks combinations of requirements)

Performance testing

- Once a system has been completely integrated
- Test for emergent properties (performance, reliability)
- To ensure that the system can process its intended load

- Usually involves running a series of tests where you increase the load until the system performance becomes unacceptable
- Concerned with
    - Demonstrating that the system meets its requirements
    - Discovering problems and defects in the system
- May have to construct an operational profile
    - A set of tests that reflect the actual mix of work that will be handled by the system
    - Not the best approach for defect testing
- Stress testing: design tests around the limits of the system (stressing the system by making demands that are outside the design limits of the software)
    - Tests the failure behavior of the system (see fail-soft)
    - Stresses the system and may cause defects to come to light that would not normally be discovered
    - Particularly relevant to distributed systems based on a network of processors (often exhibit severe degradation when loaded heavily)

User testing

- A stage in the testing process in which users/customers provide input and advice on system testing
- May involve formally testing a system that has been commissioned from an external supplier
- Could be an informal process where users experiment with a new software product to see if they like it and that it does what they need
- Essential (even when comprehensive system and release testing have been carried out)
    - Influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system
- Practically impossible to replicate the system's working environment of the customer/user
- Types of user testing
    - Alpha testing: users of software work with the development team to test the software at the developer's site
    - Beta testing: release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers
    - Acceptance testing: customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customers environment (after release testing)

Stages in acceptance testing process

- Define acceptance criteria
    - Should take place early in the process before the contract for the system is signed
    - May be difficult to define criteria early in the process
    - Detailed requirements may not be available
    - May be significant requirements change during development process

- Plan acceptance testing
    - Involves deciding on the resources, time and budget for acceptance testing and establishing a testing schedule
    - Should discuss the required coverage of the requirements and the order in which system features are tested
    - Should define risks to the testing process (system crashes, inadequate performance)
    - Discuss how these risks can be mitigated
- Derive acceptance tests
    - Tests have to be designed to check whether or not a system is acceptable
    - Should aim to test both the functional and non-functional characteristics (performance)
    - Should provide complete coverage of the system requirements
    - Difficult to establish completely objective acceptance criteria
    - Often scope for argument about whether or not a test shows that a criterion has definitely been met
- Run acceptance tests
    - Agreed acceptance tests are executed on the system
    - Should take place in the actual environment (where the system will be used)
    - User testing environment may be required to run these tests
    - Difficult to automate this process as part of the acceptance tests may involve testing the interactions between end-users and the system
    - Some training of end-users may be required
- Negotiate test results
    - Very unlikely that all of the defined acceptance tests will pass
    - Some problems will be discovered (more commonly)
    - Developer and customer have to negotiate to decide if the system is good enough to be put into use
    - Must agree on the developer's response to identified problems
- Reject/accept system
    - Involves a meeting between the developers and the customer to decide on whether or not the system should be accepted
    - Further development may be required to fix identified problems (once completed, acceptance testing phase is repeated)

Software evolution

- A system has to change if it is to remain useful (after deployment)
    - New requirements for existing software by business change and changes to user expectations
    - Modifications to correct errors being found in operation
    - To adapt it for changes to its hardware and software platform
    - To improve its performance or other non-functional characteristics
- Software evolution is important

- Organizations have invested large amounts of money and time in their software (are now completely dependent on these systems)
- Their systems are critical business assets
- They have to invest in system change to maintain the value of their assets
- Most large companies spend more time on maintenance than on development of new systems
- About 2/3 of software costs are evolution costs

- Evolution of a system can rarely be considered in isolation
    - Changes to the environment lead to system change (may trigger further environment change)
    - ‘systems-rich’ environment further increase evolution costs
- Useful software systems often have a very long lifetime
    - Large military / infrastructure systems (air traffic control systems): 30 years or more
    - Business systems: about 10 years old
- Software engineering as a spiral process with requirements, design, implementation, testing going on throughout the lifetime of the system
    - Implies: a single organization is responsible for initial software development and evolution of the software (most packaged software products)
    - Custom software: a software company develops software for a customer and the customer’s own development staff then take over the system (likely: discontinuities in the spiral process)
- Alternative view of software evolution (Rajlich and Bennet, 2000):
    - Initial development
    - Evolution: software is used successfully and there is a constant stream of proposed requirements changes (over time: structure tends to degrade, changes become more and more expensive)
    - Servicing: implementing new requirements become less and less cost effective (software still useful; but only small tactical changes; company considers software replacement)
    - Phase-out: software may still be used but no further changes are being implemented (users have to work around any problems that they discover)

Evolution processes

- Vary depending on
    - The type of software being maintained
    - The development processes used in an organization
    - The skills of the people involved
- May be an informal process: change requests mostly come from conversations between the system users and developers
- Formalized process: structured documentation produced at each stage in the process
- Driver for system evolution: system change proposals
    - From existing requirements (not yet implemented in the released version)

  - From requests for new requirements
  - From bug reports from system stakeholders
  - From new ideas for software improvement (from system development team)
  - Should be linked to the components of the system (cost and impact of the change to be assessed) (see change management)

Software evolution process:

- Change requests
- Change analysis (impact analysis)
  - Assess cost and impact of changes
  - Prototyping of proposed changes may be carried out
- Release planning
  - Possible changes: fault repair, adaptation, new functionality
  - Decision: which changes to implement in the next version of the system
- Change implementation and validation
  - An iteration of the development process
  - Revisions to the system are design, implemented and tested
  - Should modify the system specification, design and implementation to reflect changes to the system
- System release

Reasons for urgent changes:

- Serious system fault: restore normal operation to continue
- Changes to systems operating environment: unexpected affects that disrupt normal operation
- Unanticipated changes to the business running the system

Potential problematic situations in case of a handover to a separate team for evolution:

- Development team used an agile approach, but the evolution team prefers a plan-based approach
  - Missing detailed documentation to support evolution
  - May be no definitive statement of the system requirements
- Development team used a plan-based approach, but the evolution team uses an agile approach
  - Missing automated tests from agile methods
  - Code may not have been refactored and simplified as is expected in agile development
  - Some reengineering may be required to improve the code before it can be used in an agile development process

Program evolution dynamics

- The study of system change

Lehman's laws:

- Continuing change
    - A program in real-world environment must necessarily change
    - Or it becomes progressively less useful in that environment
- Increasing complexity
    - Structure of an evolving program tends to become more complex
    - Extra resources are required to preserve and simplify the structure
- Large program evolution
    - Program evolution is a self-regulating process
    - System attributes is approximately invariant for each system release (e.g. size, time between releases, number of reported errors)
- Organizational stability
    - Rate of development is approximately constant and independent of the resources devoted to system development
- Conservation of familiarity
    - The incremental change in each release is approx. constant
- Continuing growth
    - Offered functionality has to continually increate to maintain user satisfaction
- Declining quality
    - Quality of systems will decline unless they are modified to reflect changes in their operational environment
- Feedback system
    - Evolution processes incorporate multi-agent, multi-loop feedback systems
    - Treat them as feedback system to achieve significant product improvement

Software maintenance

- The general process of changing a system after it has been delivered
- Usually applied to custom software (separate development groups are involved before and after delivery)
- Different types of software maintenance
    - Fault repairs / corrective maintenance
    - Environmental adaptation / adaptive maintenance
    - Functionality addition / perfective maintenance
- Not a clear-cut distinction between these types of maintenance
- Takes up a higher proportion of IT budgets than new development (approx. 2/3)
- More of the maintenance budget is spent on implementing new requirements than on fixing bugs
- Maintenance costs are relatively high for embedded real-time systems (about 4 times higher)
- Cost effective to invest effort in designing and implementing a system to reduce the costs of future changes
- Usually more expensive to add functionality after a system is in operation than it is to implement the same functionality during development. Reasons:

- Team stability: people work for new projects
- Poor development practice: maintenance contract usually separate from system development contract
- Staff skills: maintenance staff often relatively inexperience and unfamiliar with the application domain
- Program age and structure: program's structure tends to degrade with changes

Maintenance prediction

- Maintenance costs: estimate overall maintenance costs for a system in a given time period
- System changes: predict what system changes might be proposed
- Maintainability: what parts of the system are likely to be the most difficult to maintain
- Predict number of change request (gives understanding of the relationship between the system and its external environment)
- Issues to be assess to evaluate the relationships between a system and its environment
    - Number and complexity of system interfaces
    - Number of inherently volatile system requirements
    - Business processes in which the system is used
- Sample process metrics for assessing maintainability:
    - Number of requests for corrective maintenance (decline in maintainability if more errors are being introduced into the program than are being repaired during maintenance process)
    - Average time required for impact analysis (decline in maintainability if number of program components being affected by change requests increases and this time increases)
    - Average time taken to implement a change request (decline in maintainability if amount of time needed to modify the system and its documentation increases)
    - Number of outstanding change request (decline in maintainability if the number of change requests increases)
- Model of cost estimation: COCOMO 2
  An estimate for software maintenance effort can be based on the effort
    - To understand existing code
    - To develop the new code

Software reengineering

- Reengineer legacy software systems
    - To improve their structure and understandability
    - To make maintenance easier
- May involve
    - Re-documenting the system
    - Refactoring the system architecture
    - Translating programs to a modern programming language

    - Modifying and updating the structure and values of the system's data
- Functionality of the software is not changes
- Avoid making major changes to the system architecture
- Benefits from reengineering rather than replacement
    - Reduced risk: high risk in redeveloping business-critical software (errors in system specification or development problems)
    - Reduced cost
- Activities in the reengineering process
    - Source code translation (usually automated)
    - Reverse engineering (usually automated; helps to documents its organization and functionality)
    - Program structure improvement (partially automated; analyze and improve control structure; easier to read and understand)
    - Program modularization (manual process; group together related parts; remove redundancy; may involve architectural refactoring)
    - Data reengineering (tools available; redefine database schemas; convert existing databases; clean up data; finding and correcting mistakes; removing duplicate records)
- May have to develop adapter services
    - Hide original interface of the software system and present new, better-structure interfaces
    - A legacy system wrapping process
    - An important technique for developing large-scale reusable services
- Problem of software reengineering:
    - Practical limits to how much you can improve a system be reengineering
    - Impossible to convert a system from functional to object-oriented approach
    - Major architectural changes / radical reorganizing of the system data management connect be carried out automatically
    - Less maintainable as a new system developed using modern software engineering methods

Preventative maintenance by refactoring

- Refactoring: process of making improvements to a program to show down degradation through change
    - Improve structure
    - Reduce complexity
    - Make it easier to understand
- Should not add functionality
- Should concentrate on program improvement
- Reduces problems of future change (therefore preventative maintenance)
- Reengineering:
    - After a system has been maintained for some time

- When maintenance costs are increasing
- Use of automated tools

- Refactoring
  - A continuous process of improvement throughout the development and evolution process
  - Intended to avoid structure and code degradation (otherwise increases costs and difficulties)
  - An inherent part of agile methods (extreme programming) (based around change)
  - Can be used with any approach to development
- Typical situations where refactoring can be applied ('bad smells')
  - Duplicate code
  - Long methods
  - Switch / case statements
  - Data clumping (when same group of data items reoccur in several places in a program, e.g. fields in classes, parameters in methods; can be replaced with an object encapsulating all of the data)
  - Speculative generality (when developers include generality in a program in case it is required in future; can simply be removed)
- Some methods of refactoring
  - Extract method (duplicate code)
  - Consolidate conditional expression (replace a sequence of test with a single test)
  - Pull up method (replace similar methods in subclasses with one method in a superclass)
- Effective way to reduce long-term maintenance costs of a program

Legacy system management

- For legacy systems: rather difficult to plan how to integrate evolution (system adaptation)
- Strategic options for evolving these systems
  - Scrap the system completely (when system is not making an effective contribution to business processes; changes business processes)
  - Leave the system unchanged and continue with regular maintenance (when system is still required but is fairly stable and system users make relatively few change requests)
  - Reengineer the system to improve tis maintainability (when the system quality has been degraded by changed and where new changes are still being proposed)
  - Replace all or part of the system with a new system (when factors mean that old system cannot continue in operation; when new hardware)
- Business perspective
  - Decide: business really needs the system
- Technical perspective
  - Assess: quality of the application software
  - Assess: the system's support and hardware
- 4 clusters of systems
  - Low quality, low business value (scrap the system)

  - Low quality, high business value (reengineer to improve quality; use off-the-shelf systems)
  - High quality, low business value (normal maintenance; scrap system if expensive changes become necessary)
  - High quality, high business value (normal maintenance)
- Issues for discussions on assessing the business value of a system
  - Use of the system
  - Business processes that are supported
  - System dependability
  - System outputs
- Factors used in environment assessment
  - Supplier stability (exists the supplier? Supplier likely to exists in future? If not in business, who can maintain the system?)
  - Failure rate (high rate of reported failures of the hardware? Support software crashes and forces system restarts?)
  - Age (how old is hardware and software? Significant economic and business benefits when moving to more modern systems?)
  - Performance (adequate performance? Significant effect on system users?)
  - Support requirements (local support required by hardware and software? High costs associated with support? Considering system replacement?)
  - Maintenance costs (costs of hardware maintenance and support software licenses? High annual licensing costs?)
  - Interoperability (problems interfacing the system to other systems? Use of compilers with current versions of the OS? Hardware emulation required?)
- Factors used in application assessment
  - Understandability (difficult to understand the source code?)
  - Documentation (complete, consistent, current documentation?)
  - Data (explicit data model for the system? Up-to-date and consistent data?)
  - Performance (adequate performance? Significant effect of performance problems on system users?)
  - Programming language (modern compilers available? Programming language still used for new system development?)
  - Configuration management (explicit description of the versions of components? Managed by a configuration management system?)
  - Test data (exist test data? Record of regression tests available?)
  - Personnel skills (who can maintain the application? Who have experience with the system?)
- Optional: collect data to judge the quality of the system
  - Number of system change requests
  - Number of user interfaces
  - Volume of data used by the system

Dependability and security

Sociotechnical system

- A system is more than the sum of its parts
- Software engineering: an intrinsic part of more general systems engineering
- Take a system-level view when designing software that has to be secure and dependable
- Consequences of failure are more significant
    - Extra work to contain / recover from the failure
- Critical to understand consequences of software failures to other elements in the system
- Usually the real problem: system failure rather than software failure
    - Need to examine how the software interacts with its immediate environment
- Software failures should be contained within its enclosing layers of the system stack
- Understand how faults and failure sin the non-software layers of the systems stack may affect the software
- Enterprise systems
    - Indented to help deliver a business goal

Sociotechnical system stack: layers to understand it better

- Equipment layer: hardware devices
- Operating system layer:
    - interacts with hardware
    - provides a set of common facilities for higher software layers
- Communications and data management layer:
    - Extends operating system facilities
    - Provides an interface: interaction with more extensive functionality (access to remote system, databases)
    - Middleware (layer between application and operating system)
- Application layer
    - Delivers application-specific functionality
    - Many different application programs in this layer (possibly)
    - May not be able to provide required level of privacy (need be implemented in layer below)
- Business process layer
    - Defined and enact organizational business processes
    - Make use of software system
- Organizational layer
    - Higher-level strategic processes
    - Business rules, policies, norms (should be followed when using the system)
- Social layer
    - Laws and regulations of society are defined (govern operation of the system)

Characteristics of the sociotechnical system stack:

- Most interactions between neighboring layers
- Unexpected interactions between layers (may result in problems for the system)

Systems engineering: the process of designing entire systems (not just the software in these systems)

Complex systems

- A system is a purposeful collection of interrelated components, of different kinds, which work together to achieve some objective
- Properties and behavior of the system components are inextricably intermingled
  - Successful functioning of each system component depends on the functioning of other components
- Usually hierarchical: they include other systems
  - Included systems: subsystems (can operate as independent systems)
- Systems that include software:
  - Technical computer-based systems: include hardware, software; but not procedures, processes
  - Sociotechnical systems: include one/more technical systems, people (understand purpose of the system within the system itself)
- Understand organizational environment when developing sociotechnical systems
  - To meet business needs
  - Users / managers should not reject the system

Organizational factors from system's environment affecting requirements, design and operation:

- Process changes
  - System may require changes to the work processes in the environment
  - Training will be required
  - Users might resists introduction of a system in case its changes are significant
- Job changes
  - New systems may de-skill the users in an environment
  - New systems may cause them to change the way they work
  - Users may actively resist the introduction of the system into the organization
- Organizational changes
  - System may change political power structure in an organization

Characteristics of sociotechnical systems when considering security and dependability:

- Emergent properties
  - Properties of the system as a whole
  - Depend on system components and relationships between them
  - Can only be evaluated once the system has been assembled
  - Sample: security, dependability

- Nondeterministic characteristics
  - Variable output for constant input
  - System's behavior depends on human operators

Emergent system properties

- Cannot be attributed to any specific part of the system
- They merge once the system components have been integrated
- Usually result from complex subsystem interrelationships
- Types of emergent properties
  - Functional emergent properties: when the purpose of the system only emerges after its components are integrated
  - Non-functional emergent properties: relate to the behavior of the system in its operational environment (reliability, performance, safety, security)
- Reliability from 3 perspectives:
  - Hardware reliability
  - Software reliability
  - Operator reliability
- Failures at one level can be propagated to other levels in the system
- Often difficult to anticipate how these component failures will affect other components
- Reliability of a system depends on the context in which the system is used
  - System's environment cannot be completely specified
  - Restrictions on that environment for operational systems cannot be placed by system designers
- Performance or usability
  - are hard to assess
  - Can be measured after the system is operational

Non-determinism

- A system that is not (completely) predictable
  - Hardware can be thought of as (nearly) predictable
  - People are rather unpredictable
- Sociotechnical systems are non-deterministic
  - Include people
  - Changes to the hardware, software and data in these systems are so frequent
- Interactions between changes are complex
  - Behavior of the system is unpredictable

Success criteria

- Wicked problem: no definite problem specification since the problem is so complex and involves so many related entities

-
    - Difficult to define success criteria for a system
    - Judgment of success I based on whether or not the system is effective at the time it is deployed
    - Multiple conflicting goals that are interpreted differently by different stakeholders
- Nature of security and dependability attributes (makes it sometimes even more difficult to decide if a system is successful)

Systems engineering

- Encompasses all of the activities involved in procuring, specifying, designing, implementing, validating, deploying, operating, maintaining sociotechnical system
- System engineers are concerned with
    - Software
    - Hardware
    - System's interactions with users and its environment
    - Services that the system provides
    - Constraints under which the system must be built and operated
    - Ways in which the system is used to fulfill its purpose
- Overall security and dependability of a system is influenced by activities at all stages
- Important difference between systems and software engineering
    - Involvement of a range of professional disciplines throughout the lifetime of the system (essential, because of many different aspects of complex sociotechnical systems)
- Differences between disciplines can introduce vulnerabilities into systems (compromise security and dependability of the system)
    - Different disciplines use the same words to mean different things
    - Each discipline makes assumptions about what can or can't be done by other disciplines
    - Disciplines try to protect their professional boundaries and may argue for certain design decisions because these decisions will call for their professional expertise

Overlapping stages in the lifetime of large/complex sociotechnical systems

- Procurement or acquisition
    - Decided purpose of a system
    - Established high-level system requirements
    - Made decisions on how functionality will be distributed across hardware, software and people
    - Purchased components making up the systems
- Development
    - Requirements definition
    - System design
    - Hardware and software engineering
    - System integration
    - Testing

- Operation
    - System is deployed
    - Users are trained
    - Change of planned operational processes
    - System evolves as new requirements are identified
    - System declines in value and is decommissioned and replaced

System procurement

- The initial phase of systems engineering
- Called: system acquisition
- Decisions
    - System that is to be purchased
    - System budgets and timescales
    - High-level system requirements
    - Type of system required
    - The supplier(s) of the system
    - Whether to procure a system
- Drivers for these decisions
    - State of other organizational systems (replacement system in case of a mixture of systems having troubles to communicate with each other; or maintenance problems)
    - Need to comply with external regulations (business is increasingly regulated)
    - External competition (increase effectiveness in competition; maintain competitive position)
    - Business reorganization (lead to changes in business processes that require new systems support)
    - Available budget
- Procurer deals with the contractor rather than the subcontractors (a single procurer/supplier interface)
- Decisions have a profound effect on the security and dependability of a system
- Procure a custom system: significant effort to understand and define security and dependability requirements

System procurement processes

- Define business requirements
- Survey market for existing systems
- Off-the-shelf system available
    - Adapt requirements
    - Assess existing systems
    - Choose system supplier
    - Negotiate contract
- Custom system required

- Define requirements
- Issue request to tender
- Select tender
- Negotiate contract

System development

- Goals
    - Develop / acquire all of the components of a system
    - Integrate these components to create the final system
- Requirements: the bridge between procurement and development processes
- See: waterfall model
- Plan-driven processes are used
    - Different parts of the system are being developed at the same time
- Processes of requirements development and system design are inextricably linked

System development process:

- Requirements development
    - Develop (in more detail) high-level and business requirements (identified during procurement)
- System design
    - Establish overall architecture of the system
    - Identify different system components
    - Understand relationships between them
- Subsystem engineering
    - Hardware and software components of the system are implemented
    - Configure off-the-shelf hardware and software
    - Design special-purpose hardware (if required)
    - Define operational processes for the system
    - Redesign essential business processes
- System integration
    - Put together components to create a new system
- System testing
    - An extensive, prolonged activity
    - Tune performance of the system
- System deployment
    - Make system available to its users
    - Transfer data from existing systems
    - Establish communications with other systems in the environment

Requirements and design spiral:

- Elements

  - Domain and problem understanding
  - Requirements elicitation and analysis
  - Architectural design
  - Require and assessment
- Each round of the spiral may add detail to the requirements and the design
- Some rounds may focus on requirements, some on design

System operation

- Operational processes: processes involved in using the system for its defined purpose
- Key benefit of having system operators
  - People have a unique capability of being able to respond effectively to unexpected situations
  - Use local knowledge to adapt and improve processes
  - Usually improve the process
- Actual operational processes differ from those anticipated by the system designers
- Possible problem
  - Operation of the new system alongside existing systems
  - Possibly difficult to transfer data form one system to another
- New systems may increase operator error rate

Human error

- Non-determinism
- Latent conditions
  - Contribute to system failure
  - Lead to system failure when the defenses built into the system do not trap an active failure by a system operator
  - See: Reason's Swiss cheese model of system failure
- Reduce the probability that system failure will result from human error:
  - Use different types of barriers
  - Minimize number of latent conditions in a system

Ways to view the problem of human error

- Person approach
  - Errors: responsibility of the individual and 'unsafe acts'
  - Consequence of carelessness / reckless behavior
- System approach
  - Assumption: people are fallible and will make mistakes
  - Errors: consequence of system design decisions (lead to erroneous ways of working)
  - Good systems: recognize the possibility of human error, include barriers and safeguards

System evolution

- System evolution is inherently costly:
  - Proposed changes have to be analyzed very carefully from a business and a technical perspective (changes should not simply be technically motivated)
  - Changes to subsystems may be needed (may adversely affect performance / behavior of other subsystems)
  - Reasons for original design decisions often unrecorded (evolution team have to figure out these)
  - Structure typically becomes corrupted by change (old systems)

Dependability and security

- Dependability of systems usually more important than detailed functionality
  - System failures affect a large number of people
  - Users often reject systems that are unreliable, unsafe or insecure
  - System failure costs may be enormous
  - Undependable systems may cause information loss
- Issues to consider when designing a dependable system
  - Hardware failure: mistakes in its design, manufacturing errors, natural end of life
  - Software failure: mistakes in its specification, design or implementation
  - Operational failure: human error (might be significant)
- Failures are often interrelated
- Holistic systems perspective required during design phase

Dependability properties

- Dependability: a property of the system that reflects its trustworthiness
- Trustworthiness: degree of confidence a user has that the system will operate as they expect; not fail in 'normal' use
- Principal dimensions of dependability
  - Availability: probability to be up and running and giving services at any time
  - Reliability: probability that the system is doing what is expected by users (correctness, precision, timeliness)
  - Safety: judgment of how likely the system causes damage to people / environment
  - Security: judgment of how likely the system can resist accidental/deliberate intrusions (integrity, confidentiality)
- Other system properties (dependability properties)
  - Reparability
  - Maintainability
  - Survivability
  - Error tolerance
- Things to ensure when developing dependable software
  - Avoid introduction of accidental errors into the system during software specification and development

  - Design verification and validation processes being effective in discovering residual errors that affect the dependability of the system
  - Design protection mechanism that guard against external attacks that can compromise the availability or security of the system
  - Configure the deployed system and its supporting software correctly for tis operating environment
- Need for fault tolerance
  - Need to include redundant code to help them monitor themselves, detect erroneous stats and recover from faults before failure occurs
- Increases development costs
  - Extra design, implementation, validation costs
  - High validation costs for ultra-dependable systems (safety-critical control systems)
  - Testing

Availability and reliability

- Closely related properties
- Can be expressed as numerical probabilities
- Reliability: probability of failure-free operation over a specified time, in a given environment, for a specific purpose
- Availability: probability that a system, at a point in time, will be operational and able to deliver the requested service
- Standard definitions do not take into account
  - The severity of failure
  - The consequences of unavailability
- Users often accept minor failures
- Users rarely accept failures with high consequential costs
- Reliability terminology
  - Human error or mistake (see above)
  - System fault: characteristic of a system that can lead to a system error
  - System error: erroneous system state that can lead to system behavior that is unexpected by system users
  - System failure: event that occurs at some point in time when the system does not deliver a service as expected by its users
- Practical reliability of a program depends on
  - The number of inputs causing erroneous outputs during normal use
- Practical reliability does not really depend on
  - Software faults occurring in exceptional situations (do not always result in system errors)
- Complementary approaches used to improve the reliability of a system
  - Fault avoidance: avoid error-prone programming language constructs (pointers), use static analysis to detect program anomalies

- Fault detection and removal: use of verification and validation techniques (systematic testing and debugging)
- Fault tolerance: incorporate self-checking facilities in a system, use redundant system modules (techniques that ensure that faults in a system do not result in system errors or vice versa)

Safety

- Safety-critical system:
    - Essential that system operation is always safe
    - System should never damage people or system's environment
    - Sample: control- and monitoring systems in aircraft, process control system in chemical and pharmaceutical plants, automobile control systems
- Software control is essential
    - Need to manage large numbers of sensors and actuators with complex control laws
    - Sample: advanced, aerodynamically unstable, military aircraft
- Classes of safety-critical software
    - Primary safety-critical software: embedded as a controller in a system (malfunctioning cause hardware malfunction causing human injury / environmental damage)
    - Secondary safety-critical software: can indirectly result in an injury (computer-aided engineering design system, mental health care management systems)
- Reasons why reliable software systems are not necessarily safe
    - No certainty that a system is fault-free and fault-tolerant
    - Specification may be incomplete (no description of required behavior of the system in some critical situations)
    - Hardware malfunctions (causing the system to behave in an unpredictable way)
- Safety terminology
    - Accident / mishap: an unplanned event or sequence of events resulting in human death / injury, damage to property or environment
    - Hazard: condition with the potential for causing or contributing to an accident (failure of sensors)
    - Damage: measure of the loss resulting from a mishap (can range from many people being killed to minor injury or property damage)
    - Hazard severity: assessment of the worst possible damage that could result from a particular hazard (range from catastrophic to minor)
    - Hazard probability: probability of the events occurring which create a hazard (tend to be arbitrary, range from probably to implausible)
    - Risk: a measure of the probability that the system will cause an accident
- How to assure safety or reduce consequences of accidents
    - Hazard avoidance
    - Hazard detection and removal
    - Damage limitation

Security

- A system attribute
- Reflects the ability of the system to protect itself from external attacks (may be accidental or deliberate)
- Security terminology
    - Asset: something of value which has to be protected (software system, data used by that system)
    - Exposure: possible loss or harm to a computing system (loss/damage to data, loss of time and effort if recovery is necessary after a security breach)
    - Vulnerability: weakness in a computer-based system that may be exploited to cause loss or harm
    - Attack: exploitation of a system's vulnerability
    - Threats: circumstances that have potential to cause loss or harm (see system vulnerability that is subjected to an attack)
    - Control: protective measure that reduces a system's vulnerability (e.g.: encryption)
- Samples:
    - Military system
    - Systems for electronic commerce
    - Systems that involve the processing and interchange of confidential information
- Main types of security threats in any networked system
    - Threats to the confidentiality of the system and its data
    - Threats to the integrity of the system and its data (can damage or corrupt software / data)
    - Threats to the availability of the system and its data (can restrict access to the software / data for authorized users)
- Sample security threats created by human failings
    - Easy-to-guess passwords, writing down passwords
    - System administrators: setting up access control or configuration files, users don't install protection software
- Controls to enhance system security
    - Vulnerability avoidance
    - Attack detection and neutralization
    - Exposure limitation and recovery

Dependability and security specification

- Attention to detail when system requirements are derived

Risk-driven requirements specification

- Dependability and security requirements: protection requirements
    - How system should protect itself from internal faults
    - Stop system failures
    - Stop accidents or attacks

    - Facilitate recovery
- Risk-driven approach to requirements specification takes into account
    - Dangerous events that may occur
    - Probability that these will actually occur
    - Probability that damage will result from such events
    - Extent of the damage caused
- Analyze possible causes of dangerous events
- Establish security and dependability requirements
- Stages in a general risk-driven specification process
    - Risk identification: interactions between systems, care conditions in its operating environment (risk description)
    - Risk analysis and classification: each risk is considered separately; serious/implausible risks are selected for further analysis (risk assessment)
    - Risk decomposition: discover potential root causes for selected risk (root cause analysis)
    - Risk reduction: proposals to reduce/eliminate identified risks (dependability requirements)
- Phases of risk analysis (for large systems)
    - Preliminary risk analysis: identify major risks from the system's environment
    - Life-cycle risk analysis: concerned with risks arising from system design decisions (during system development)
    - Operational risk analysis: concerned with system user interface and risks from operator errors

Safety specification

- Safety-critical system:
    - Failures may affect the environment of the system
    - Failures may cause injury/death to people in that environment
- Principal concern: identify requirements that will minimize the probability that such system failures will occur
- Safety requirements
    - Protection requirements
    - Not concerned with normal system operation
    - May specify system shutdown to maintain safety
- Avoid overprotection
- Find balance between safety and functionality
- Safety specification process
    - Risk identification
    - Risk analysis
    - Risk decomposition
    - Risk reduction

Hazard identification

- Can lead to an accident
- Consider different types of hazards (physical, electrical, biological, radiation hazards, service failure hazards etc.)
- Identify hazards
    - From previous experience
    - From an analysis of the application domain
    - Group working techniques (brainstorming)
- Software-related hazards are normally concerned with
    - Failure to deliver a system service
    - Failure of monitoring and protection systems (included in a device; e.g.: low battery levels)

Hazard assessment

- Focuses on
    - Understanding the probability that a hazard will occur
    - The consequences of an accident / incident associated with that hazard
- Understand seriousness of a given hazard
- Risk categories to be used in hazard assessment
    - Intolerable risks: threaten human life
    - As low as reasonably practical risks (ALARP): less serious consequence; serious but very low probability of occurrence
    - Acceptable risk: minor damage (possibly reduce all of them)
- Boundaries depend on social and political factors
- Hazard assessment involves estimating hazard probability and risk severity (usually difficult)

Hazard analysis

- Process of discovering the root causes of hazards in a safety-critical system
- Use of top-down (deductive) or bottom-up approach
- Techniques
    - Hazard decomposition / analysis
    - Reviews and checklists
    - Formal techniques
    - Formal logic
    - Fault tree analysis
- Fault trees are also used to identify potential hardware problems

Risk reduction

- Derive safety requirements from identified potential risks that manage the risks and ensure that incidents / accidents do not occur
- Possible strategies

 - Hazard avoidance: system is design that hazards cannot occur
 - Hazard detection and removal: hazards are detected and neutralized before they result in an accident
 - Damage limitation: system is design so that consequences of an accident are minimized
- Possible solutions
 - Arithmetic error: an arithmetic computation causes a representation failure
 - Algorithmic error: more difficult situation as there is no clear program exception that must be handled

Reliability specification

- Reliability of a system depends on
 - Hardware reliability
 - Software reliability
 - Reliability of system operators
- Reliability: a measurable system attribute
- Reliability requirements are of two kinds
 - Non-functional requirements: number of failures (quantitative reliability requirements)
 - Functional requirements: system/software functions to avoid/detect/tolerate faults (qualitative reliability requirements)
- Process of reliability specification can be based on general risk-driven specification processes
 - Risk identification (identify types of system failures)
 - Risk analysis (estimate cost and consequences)
 - Risk decomposition (root cause analysis)
 - Risk reduction (generate quantitative reliability specifications)

Reliability metrics

- Important metrics used to specify reliability plus an additional metric (used to specify related system attribute of availability)
 - Probability of failure on demand (POFOD): define the probability that a demand for service from a system will result in a system failure
 - Rate of occurrence of failures (ROCOF): probable number of system failures being likely to be observed relative to a certain time period
 - Availability (AVAIL): reflects a system's ability to deliver services when requested
- To assess the reliability of a system
 - Capture data about its operation (number of system failures; time/number of transactions between system failures plus total elapsed time of transactions; repair/restart time after system failure)
- Use calendar time for system being in continuous operation (monitoring systems)

Non-functional reliability requirements

- Quantitative specification of the required reliability and availability of a system

- Calculated using one of the metrics described above
- Advantages in deriving quantitative reliability specifications
    - Process of deciding what required level of reliability helps to clarify what stakeholders really need
    - Provides a basis for assessing when to stop testing a system
    - A means for assessing different design strategies intended to improve the reliability of a system
- Consider associated losses that could result from a system failure
- Problem of specifying reliability using metrics
    - Possible to over-specify reliability
    - Incur high development and validation costs
- Organizations must be realistic about whether it is worth specifying and validating a very high level of reliability
- Steps to take to avoid over-specification of system reliability
    - Specify availability and reliability requirements for different types of failures
    - Specify availability and reliability requirements for different services separately
    - Decide whether high reliability in a software system is really needed

Functional reliability specification

- Involves identifying requirements that define constraints and features that contribute to system reliability
- Types of functional reliability requirements for a system
    - Checking requirements (identify checks on inputs to the system to ensure that incorrect / out-of-range inputs are detected before being processed by the system)
    - Recovery requirements (help the system recover after a failure has occurred)
    - Redundancy requirements (specify redundant features of the system that ensure that a single component failure does not lead to a complete loss of service)
- May include process requirements for reliability
- No simple rules for deriving functional reliability requirements

Security specification

- Some commonalities with safety requirements
- Impractical to specify them quantitatively
- Security is a more challenging problem than safety.
- Security requirements need to be more extensive than safety requirements
- Possible security requirements
    - Identification requirements (whether to identify users before interaction)
    - Authentication requirements (how users are identified)
    - Authorization requirements (privileges and access permissions of identified users)
    - Immunity requirements (show a system protects itself against viruses, worms and similar threats)

- Integrity requirements (how to avoid data corruption)
- Intrusion detection requirements (mechanisms on how to detect attacks on the system)
- Non-repudiation requirements (a party in a transaction cannot deny its involvement in that transaction)
- Privacy requirements (how to maintain data privacy)
- Security auditing requirements (how to audit and check system use)
- System maintenance security requirements (how to prevent authorized changes from accidentally defeating security mechanisms)

- Stages in risk analysis and assessment process (see above)
  - Preliminary risk analysis
  - Life-cycle risk analysis
  - Operational risk analysis
- Process stages for security requirements
  - Assess identification
  - Asset value assessment
  - Exposure assessment
  - Threat identification
  - Attack assessment
  - Control identification
  - Feasibility assessment
  - Security requirements definition

Formal specification

- Mathematically-based approaches to software development where you define a formal model of the software
- Eliminate software failure by proving that a program is consistent with a given model
- Usually developed as part of a plan-based software process
- Not just essential for a verification of the design and implementation of software
- Automated support for analyzing a formal specification has been developed
- Formal specification in a plan-based software process
  - User requirements definition
  - System requirements specification
  - Architectural design
  - Formal specification
  - High-level design
- Arguments against formal system specification
  - Problems owners and domain experts cannot understand a formal specification (cannot check that it accurately represents their requirements)
  - Difficult to estimate the possible cost savings that will result from its use
  - Most software engineers have not been trained to use formal specification languages
  - Difficult to scale current approaches to formal specification up to very large systems
  - Formal specification is not compatible with agile methods of development

Dependability engineering

- Critical systems have a high level of dependability
    - Process control systems, protection systems, medical systems, telecommunications switches, flight control systems
- Special development tools and techniques required to enhance dependability
- Concerned with the techniques being used to enhance dependability of both critical and non-critical systems
- Complementary approached being supported by these techniques
    - Fault avoidance (approaches to avoid design and programming errors)
    - Fault detection and correction (verification and validation processes discover and remove faults in a program before being deployed)
    - Fault tolerance (faults are detected at run-time and are managed to prevent system failure)
- Software development companies accept that their software will always contain some residual faults
- Must product the evidence that can convince a regulator that the system is dependable

Redundancy and diversity

- Fundamental strategies for enhancing the dependability of any type of system
- Redundancy: spare capacity is included in a system that can be used if part of that system fails
- Diversity: redundant components of the system are of different types