Software Engineering Principles

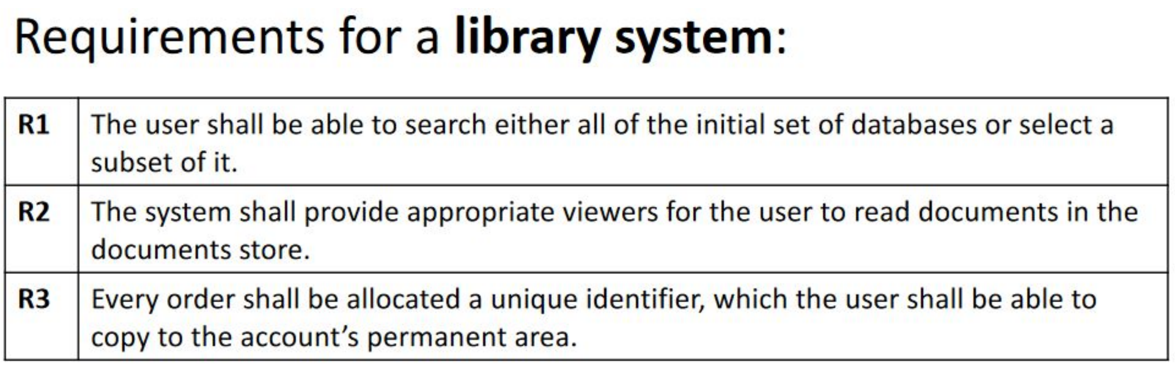
1. Software Development & Project Management

2. Requirements Description and Elicitation

- **Elicitation meaning**: *any of various data collection techniques in social sciences or other fields to gather knowledge or information from people*

- Intro:
  - Why spend time on requirements?
  - What is a requirements specification? And who uses them?
    - Managers?
    - Customers?
    - Quality assurance?
    - software designers (arkitekt)?
    - Coders?

  - Quick example:



Requirements for a **library system**:

R1	The user shall be able to search either all of the initial set of databases or select a subset of it.
R2	The system shall provide appropriate viewers for the user to read documents in the documents store.
R3	Every order shall be allocated a unique identifier, which the user shall be able to copy to the account's permanent area.

    - Can the above be improved?

  - What are good requirement specifications?
    - **Correct -** Specifying something actually needed
    - **Unambiguous -** Only one interpretation
    - **Complete -** Includes all significant requirements
    - **Consistent -** No requirements conflict
    - **Verifiable -** All requirements can be verified
    - **Modifiable -** Changes can easily be made to the requirements
    - **Traceable -** The origin of each requirement is clear

- REACTIVE SYSTEMS

    - Interactive
    - Interrupt-driven
    - non-terminating
    - State-dependent response
    - Real-time
    - Examples:
        - Workflow systems, embedded software, web market places

- Transformational systems
    - ***compute output from an input and then terminate***
    - Terminating
    - Sometimes interactive
    - Not interrupt-driven
    - Output not state-dependent
    - Output defined in terms of input
    - Sequential
    - Usually not real-time
    - Examples:
        - compiler, web-shop, search algorithm, LaTex, etc.

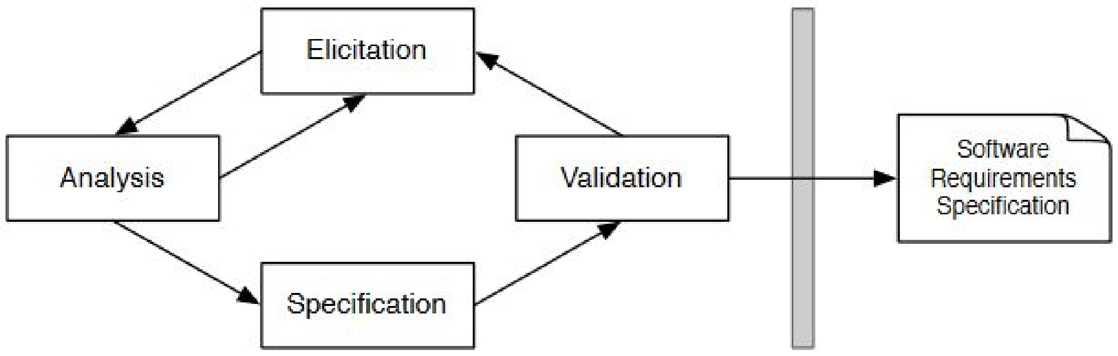
- **Non-functional** requirements
    - Often described as "**Quality of Service**"
    - Performance, Usability, Security, Maintainability, Reliability, Efficiency
- **Functional** requirements
    - What the system should do
    - These are the requirements that the end user specifically demands as basic facilities that the system should offer.
- **Behavioural** requirements **TODO**
    - Only describes the behaviour of the system
- **Domain requirements TODO**
    - Requirements to a domain, and are not linked to a specific user.
- **Structural** requirements **TODO**
    - Describes constraints on the internal composition of the system
- **Goal level req TODO**
- **Domain level req TODO**
- **Product level req TODO**
- **design level req TODO**

- For guidelines on how to create good software requirement specification:
    - **Outline for SRS IEEE**
        - **1. Introduction**
            - Purpose of the doc
            - intended audience
            - Scope
            - Summerize what the product will do
            - objectives and goals for the system
            - Word list: Definitions, acronyms, abbreviations

        - references
        - overview
    - **2. Overall description**
        - Perspective: Show diffs between proposed system, and existing systems.
        - Diagrams showing the major components of the larger system.
        - Summary of the major system functions. Should be organized i a readable list for the customer or others
        - Explain the relation between the functions.
        - Intended user for the product: What level of expertise should the user have.
        - Constraints for developers: hardware, language etc.
        - Any assumptions that may affect the SRS
    - **3. Specific requirements (the detailed ones)**
        - contains requirements to a level of detail sufficient to enable designers to design a system to satisfy those requirements, and testers to test that the system satisfies those requirements
        - Specifies system interfaces, user interfaces, hardware interfaces, software interfaces, communications interfaces.
        - A detailed description of all inputs and outputs from the software system should be given
        - Functional requirements
            - validity checks on input; exact sequence of operations; responses to abnormal situations; relationship of outputs to inputs
            - Task descriptions, use-case diagrams, scenarios, activity diagrams can be used here
        - None functional reqs
            - Performance requirements › Speed, availability, response time
            - security, portability, reliability
            - should be specified in measurable terms
            - Design constraints
        - Domain requirements
            - Explain the application domain and constraints on the application domain
    - **4. Appendices**
        - Any other important material
    - IMPORTANT
        - Specify what the system should do – not how

- Difference between Use case and Task description

- Requirements process

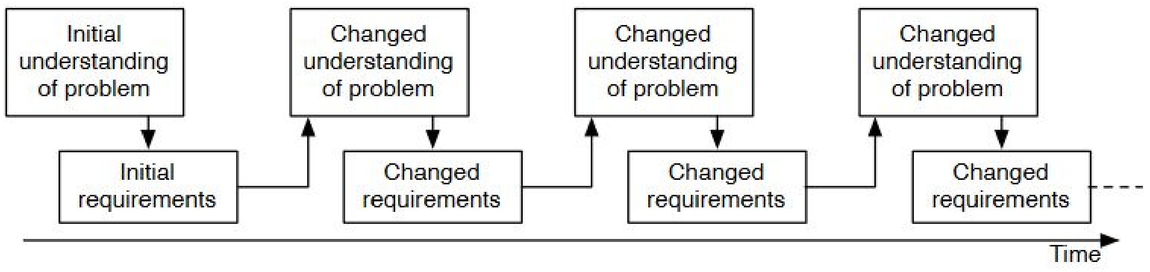


  - Elicitation: Collecting the users requirements
  - Analysis: Understanding and modelling of desired behaviour
  - Specification: Documenting the behaviour of the proposed software system
  - Validation: Checking that the specification matches the users requirements

- Requirements validation
  - Overall: demonstrate that the specification meets the requirements
  - Check requirements and specification against:
    - Correctness, Consistency, Completeness, Feasibility, Relevance, Testability, Traceability
  - Different validation techniques, e.g.
    - Walkthrough: author presents requirements to stakeholders
    - Review: Stakeholders examine requirements individually and discuss their findings
    - Simulation and production of prototypes
    - Model checking
    - Mathematical proof

- Problems with requirements
  - In particular natural language requirements suffer from:
    - Lack of clarity: natural language can be imprecise and ambiguous
    - Requirements confusion: failure to distinguish functional requirements from quality requirements and constraints
    - Requirements amalgamation: different requirements stated as a single requirement

- Requirements evolution


Initial understanding of problem
Initial requirements
Changed understanding of problem
Changed requirements
Changed understanding of problem
Changed requirements
Changed understanding of problem
Changed requirements
Time

    - Not all requirements change
        - Enduring requirements:
            - stable requirements describing core functionality
        - Volatile requirements:
            - requirements that are likely to change

- REQUIREMENTS ANALYSIS
    - Make reqs testable
    - Priotise reqs via “Essential, desirable, optional” (Mosscov)
    - Two kinds of requirements document are produced
        - (User) Requirements (Goal/Domain)
        - (Software) Specification (Product/Design):
    - Modelling notations
        - Problem Frames (graphical notation: requirements) viaframes
        - Task Descriptions (structured text: requirements)
        - UML / SysML (graphical notation: functionality)
        - VDM (formal notation)

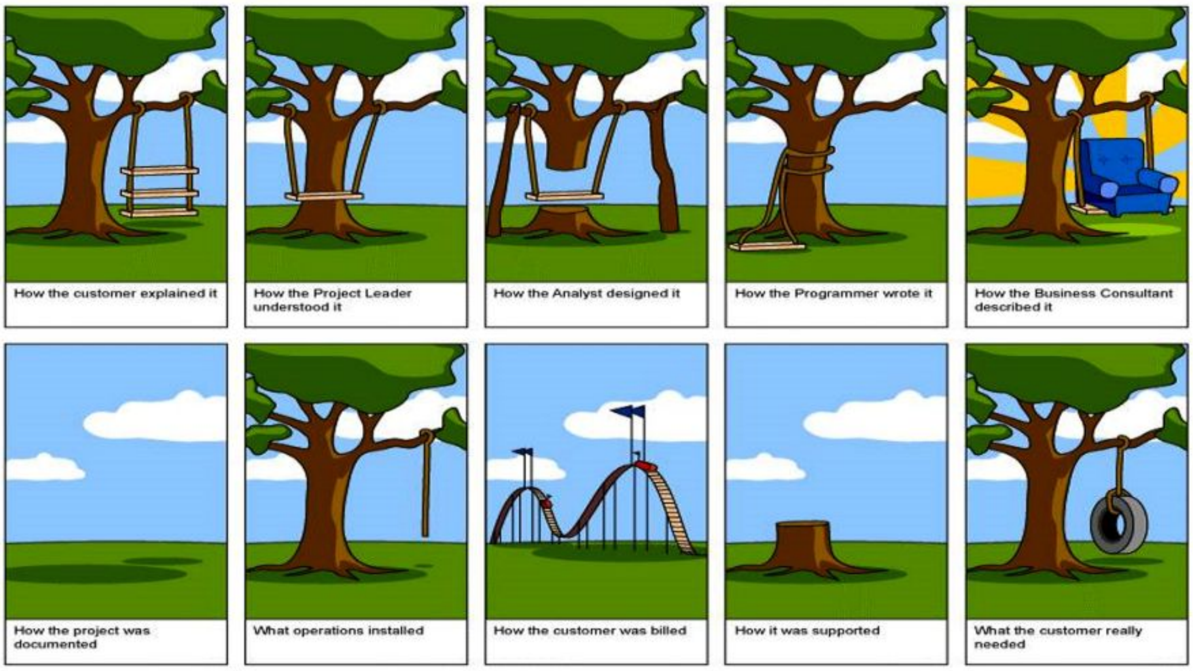
- REQUIREMENTS VALIDATION
    - Check for : Correctness Consistency Unambiguity Completeness Feasibility Relevance Testability Traceability
    - Walkthrough: Author presents requirements to stakeholders
    - Review: Stakeholders examine requirements individually and discuss their findings
    - Others: Simulation and production of prototypes, Model checking, Mathematical proof

- REQUIREMENTS MANAGEMENT
    - Reqs can change:
        - Wrong balance of compromises between requirements
        - Diverse user community with different needs and priorities
        - Users (not stakeholders) during requirements capturing
        - Changing business priorities
        - New hardware may be introduced
        - Changing legislation or regulations
    - Since reqs change, we the versions must be traceable (sematic versioning, git etc.)
    - Comparing proposals

        - calculate a total sum of points from different aspects and check for the best.

- ELICITATION
    - Finding and formulating reqs for the system.
        - Formulating overall goals of new system (Mission statement)
        - Describing the users current work process and its problems.
        - Detailed description of issues the system must solve
        - Give possible solutions to the problems
        - Turn issues and possibilities into reqs
            - These steps are iterative, and may change.

- ELICITATION - why is it hard?
    - Stakeholders cannot express what they want
    - Hard for users to explain their daily tasks
    - Hard to imagine new ways of doing tasks
        - and consequences for this
    - Different stakeholders have conflicting views
    - General resistance to change
    - Too many “nice to have” requirements are specified
    - Changes spawn new requirements


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

- Stakeholders:
    - A paying customer
    - Users of the current system
    - Domain experts
    - Market researchers
    - Lawyers or auditors legislation
    - Software engineers

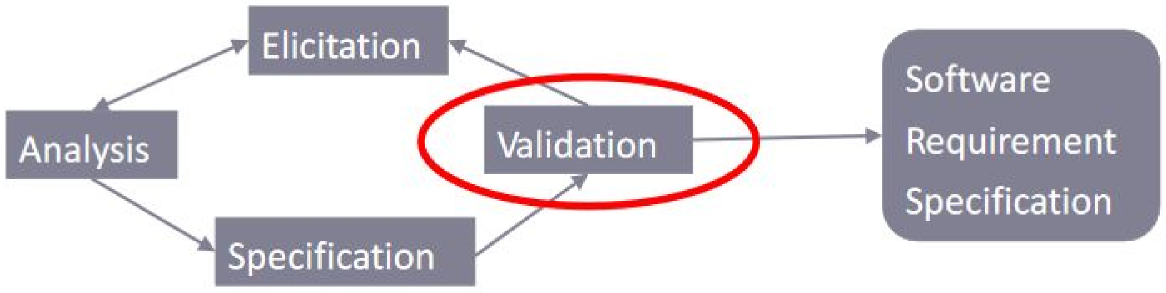
- Activities for getting reqs
    - Requirements gathered by stakeholders
    - Interview stakeholders
        - Current work process and problems are identified
        - Broad interview with many different users
        - Use open questions → more open discussion
        - Agree with users on scenarios / use-cases
        - Quantitative(Questionnaires)
            - Larger group of users
            - Hard to analyse results – cannot ask users additional questions
            - Easy to misunderstand answers
        - Qualitative (interviews)
            - Allows follow up questions
            - Easier to understand
    - Review available documentation
    - Observe current system (We were told that this was best)
        - Hard to describe what is done – easier to observe work process
        - People often miss out on simple important stuff.
            - Usability tools when observing:
            - Think-out-loud
            - Measure time used
            - Measure errors made
            - Count number of keystrokes
    - Apprenticing with users
    - Brainstorming with current and potential users
        - Stakeholder meetings can form focus group
            - More structured than a brainstorm session
            - Problems and issues
            - Ideal solutions
- What should we log from these activities:
        - A description of the present work in the domain
        - A list of the present problems in the domain
        - future goals and critical issues

- User involvement
    - Members of design teams
    - Test users
    - Reviewers
    - and so on…

- Stakeholder analysis
    - Who are the stakeholders?
    - What are their goals?
    - Which risks and costs do they see?
    - Made in large, small or 1-on-1 meetings

3. Requirements Validation

- If you get this subject just esially : http://dropud.nu
- Where we are in requirements description and elicitation, looked at analysis and elicitation, we now look at how we validate requirements.



- Validation: Check that the specification matches the users requirements
  - Review and Inspection
    - Informal Review
    - Review Meeting
    - Formal Inspections
    - How to inspect

- (informal) reviews
  - Everyone makes mistakes
  - Create open atmosphere (increase productivity)
  - Find errors in program early (before it is run the first time)
  - Find quality issues
  - improve programming skills of all involved

- Review meeting
  - **Purpose:** to evaluate a software product to
    - determine its suitability for its intended use
    - identify discrepancies from specifications and standards
  - Participants read documents in advance
    - Then bring their comments to a meeting for discussion
  - A review
    - May provide recommendations and suggest alternatives
    - may be held at any time during a project
    - Need not reach conclusions on all points
  - What should not happen in a review?
    - Improvements to the program
    - Blaming programmers
    - Finger pointing

- (Formal) Inspections
  - **Purpose:** detect and identify software product anomalies by systematic peer evaluation
  - The inspection leader is not the author
    - is a trained moderator

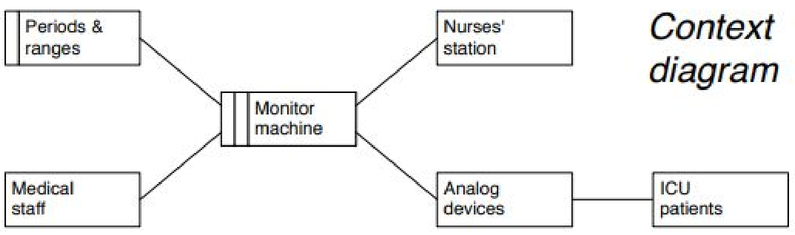
        - Organizes the selection of inspectors
        - distributes the documents
        - leads the meeting
        - ensures all follow up actions are taken

- How to inspect?
    - Set an agenda and maintain it
    - Limit debate and rebuttal
    - Dot not attempt to solve every problem noted
    - Take written notes
    - Insist on advance preparation
    - Conduct meaningful training for all participants
    - Inspect your earlier inspections

- Problem Frames
    - Patient monitoring problem
    - Context diagram
    - Domain interfaces
    - Problem Diagrams
    - Frame Concern

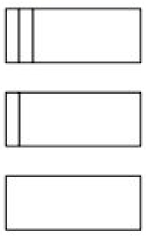
- Problem Frames: wth is it
    - Focus on the problem solving aspect of requirements modelling
    - Do not commit early to solutions: do not implement before you understand the problem
    - Abstract from the state and behaviour
    - Context Diagram

Patient Monitoring Problem



Context diagram

Problem domains:



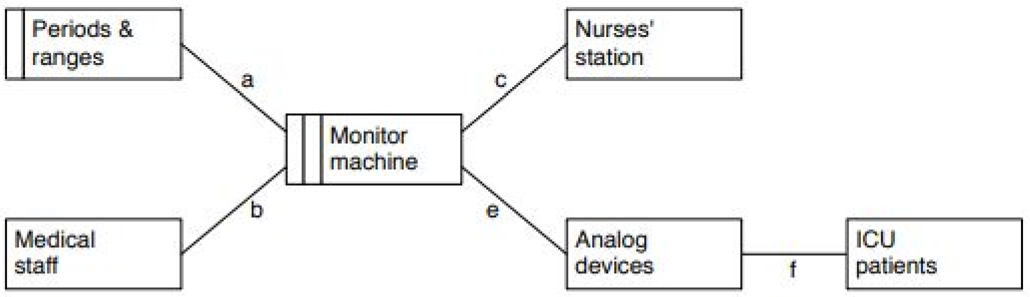
the *machine domain* is the computer for which the software is to be developed

a *designed domain* is the physical representation of some information, e.g., on a hard disk or screen

a *given domain* is a problem domain whose properties are given. The domain cannot be designed.

- Domain Interfaces

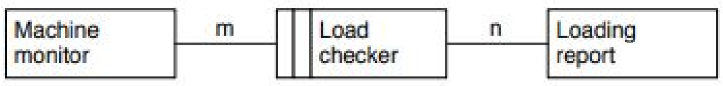
Patient Monitoring Problem: Domain Interfaces



a:	Period, Range, PatientName, Factor	c:	Notify
b:	EnterPeriod, EnterRange, EnterPatientName, EnterFactor	e:	RegisterValue
		f:	FactorEvidence

Each interface is a set of *shared phenomena*.

A phenomenon is an *event*, *state* or *value*.



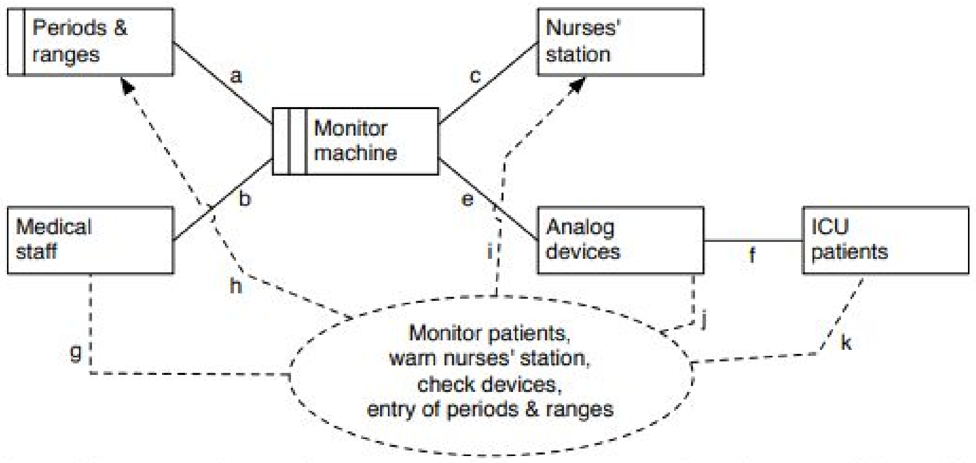
m: IdleState
n: WriteReportInfo

Computers can also be given domains.
A machine domain of one problem can be a given domain of another.

E.g. a problem of analysing whether the monitor machine of the patient monitoring problem is overloaded.

- Problem Diagrams
  - A problem diagram shows how requirements are related to proble domains

Patient Monitoring Problem: Problem Diagrams

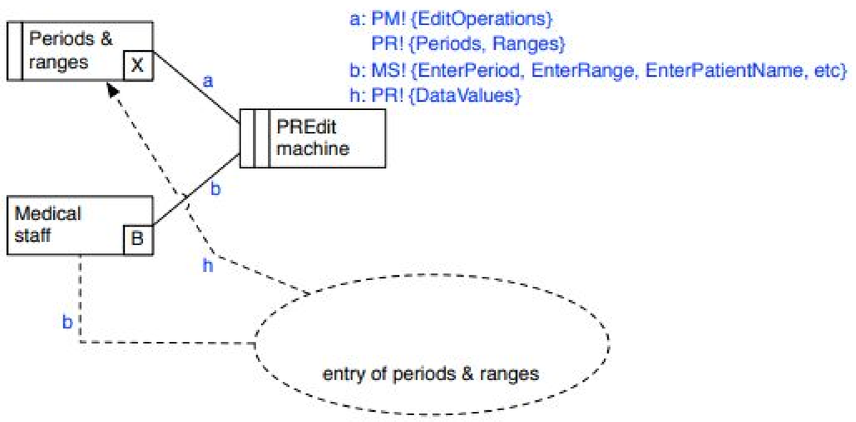


A *problem diagram* shows how requirements are related to problem domains.

◌	a *requirement* to be respected
- - - - - -	a *requirement reference*: the requirement refers to phenomena of the connected domain
- - - - - -▶	a *constraining requirement reference* the requirement constrains phenomena of the connected domain

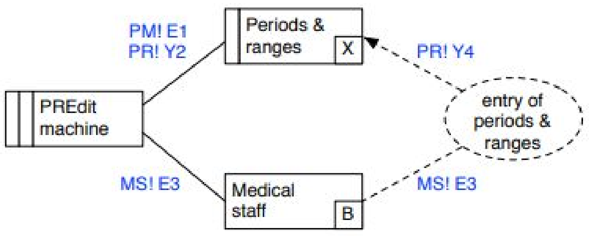
- ○ If a problem diagram is too complex: You should project the diagram into smaller diagrams
- ○ Then you're able to further describe the diagram

Patient Monitoring Problem: Subproblem Diagram

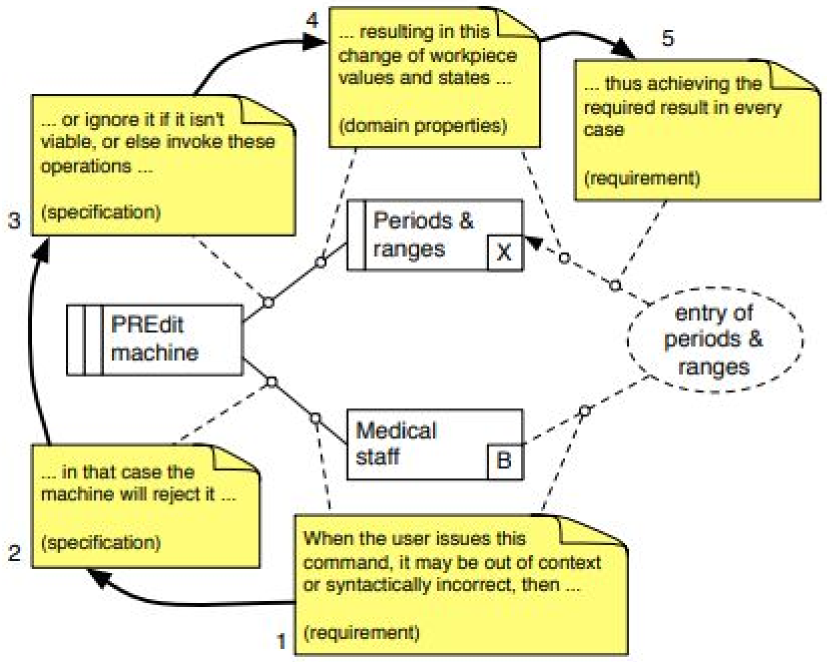


$\boxed{X}$	a *lexical domain*: physical representation of data
$\boxed{C}$	a *causal domain*: predicable relationship between phenomena, e.g., a motor
$\boxed{B}$	a *biddable domain*: usually consists of people, no predictable internal causality

- Frame Concern
    - Transform the problem diagram into a known and recognizable class



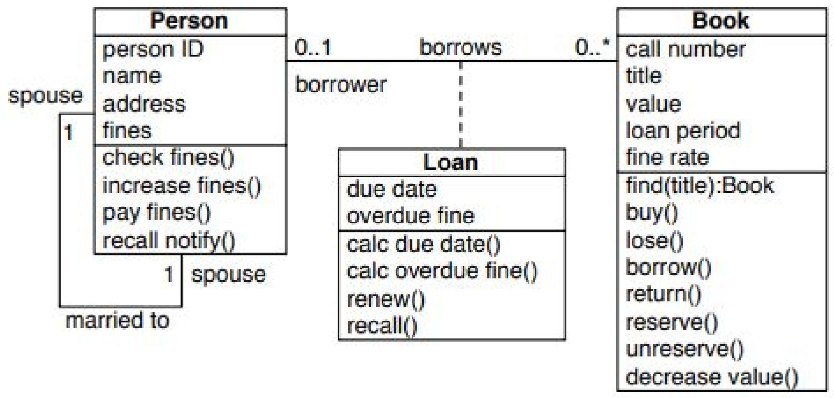
    - This frame captures and defines a commonly found class of simple subproblem. Then you put frame concern on the problem frame


4
... resulting in this change of workpiece values and states ... (domain properties)
5
... thus achieving the required result in every case (requirement)
... or ignore it if it isn't viable, or else invoke these operations ... (specification)
3
Periods & ranges
X
PREdit machine
entry of periods & ranges
Medical staff
B
... in that case the machine will reject it ... (specification)
2
When the user issues this command, it may be out of context or syntactically incorrect, then ... (requirement)
1

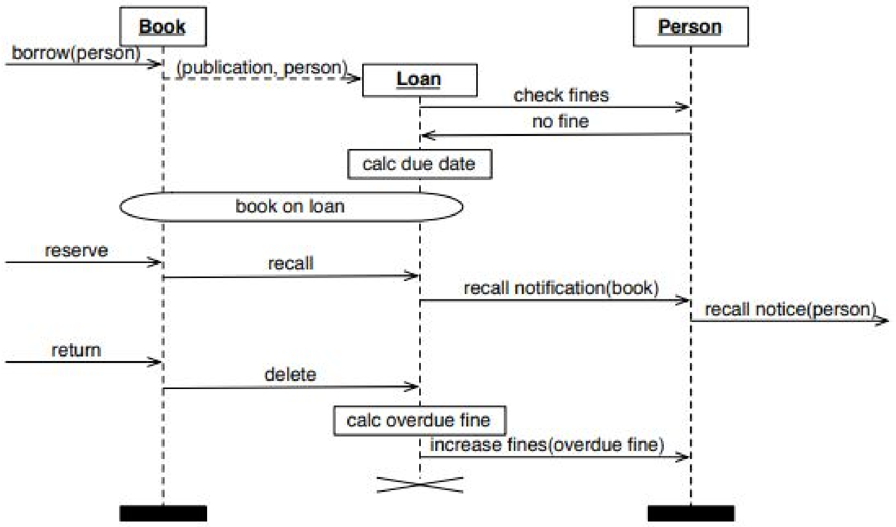
    - Problems working wit problem frames
        - Overrun, initialization, reliability, identities and completeness
    - Composition concerns
        - Commensurable descriptions, consistency, precedence, interference and synchronization

- UML
    - Collection of notations used to document software specifications and designs
    - Oriented towards implementation on a computer
    - We look at
        - Class diagrams
        - Message sequence charts
        - State chart diagrams
        - Use cases
        - The object constraint langauge

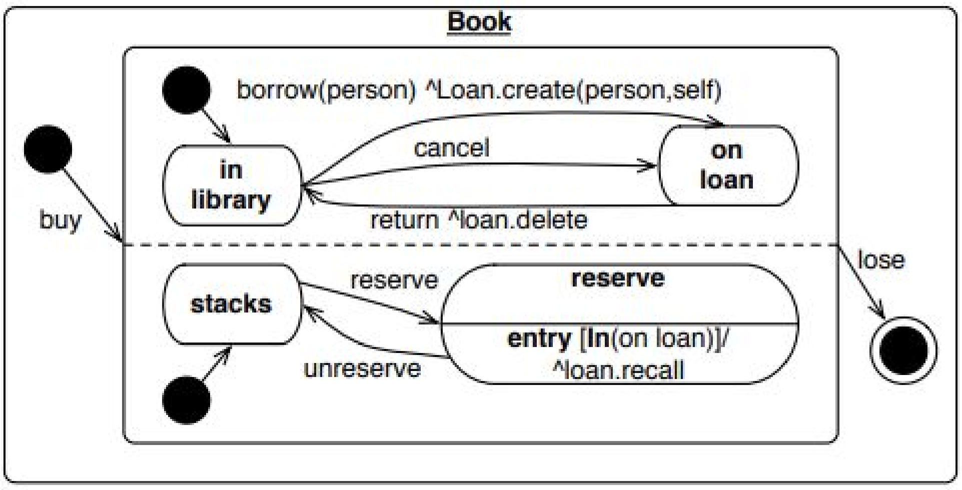
Library: Class Diagram



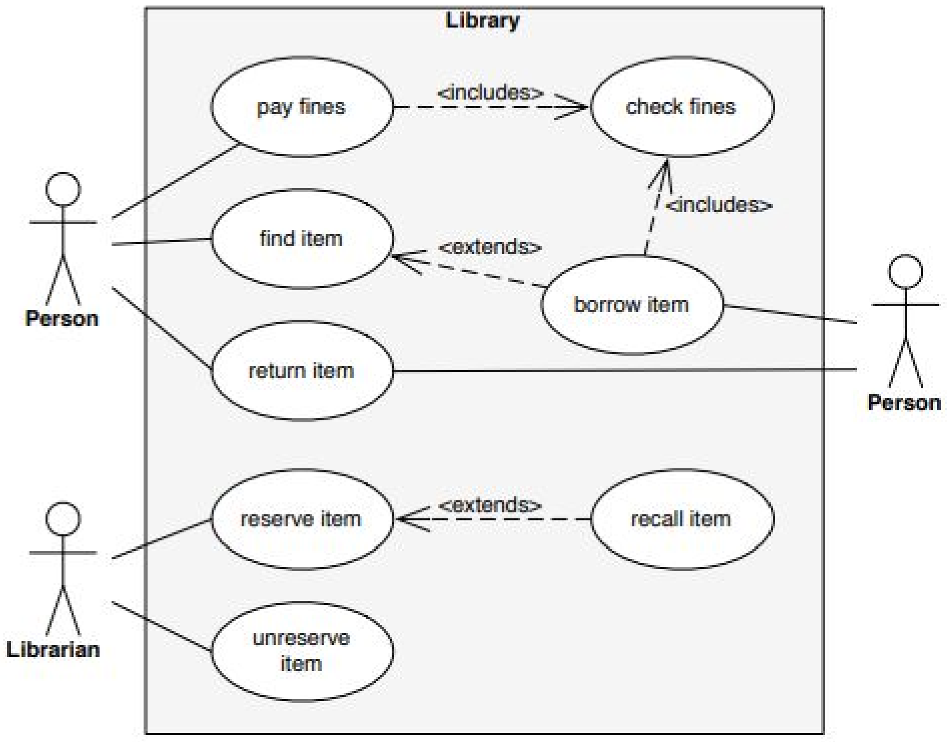
Library: Message Sequence Chart



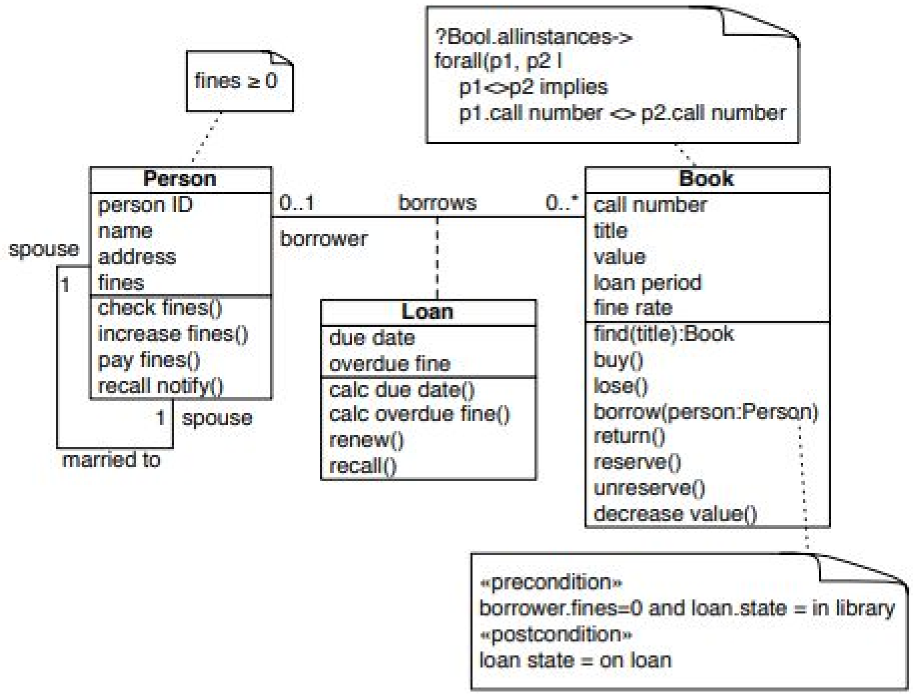
Library: State Chart Diagrams



Library: Use Cases



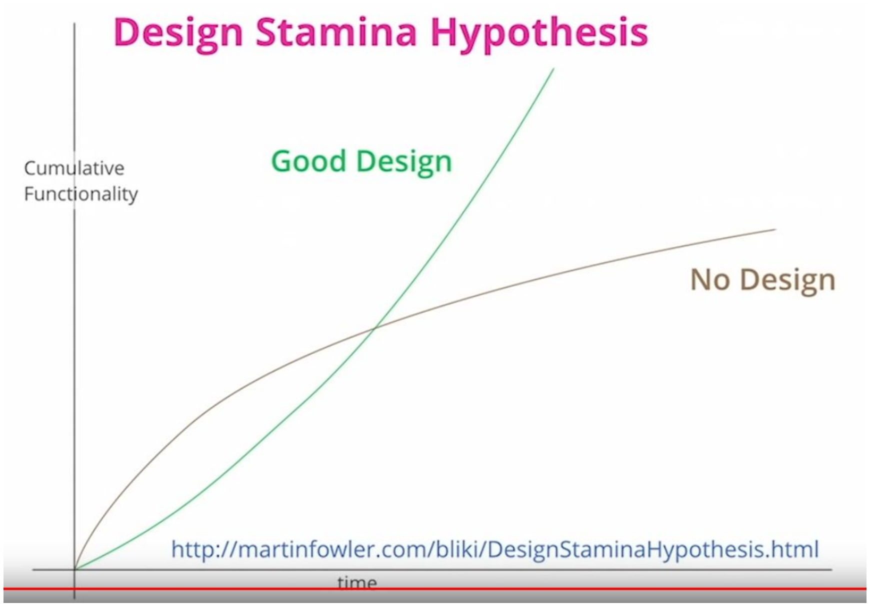
Library: Object Constraint Language



Where to go about after this if you have enough time?

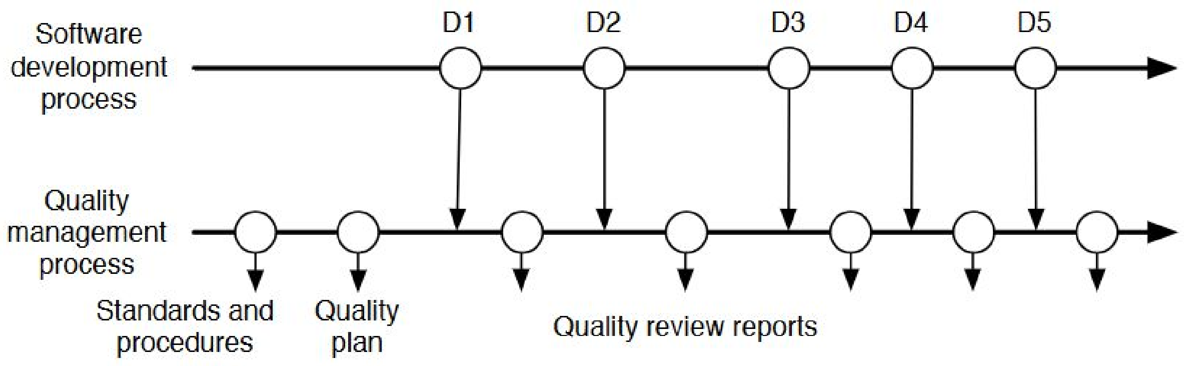
4. Software Quality

- What is Software Quality?
  - Simple (but bad answer): Software should meet its specification
  - What about about:
    - Usaability?
    - Efficiency?
    - Reliability
    - Maintainability?
    - Portability?
    - Reusability?
- So why is quality important?
  - As Martin fowler explains:
  - We may start with having fast production code, however, in the long run. Every new feature will result in more bugs and harder to debug code.



- So how can good quality be ensured?
  - Quality assurance
  - Quality control

- Quality management must occur continually



- A Lot of different techniques of validation and verification

    - Testing
    - Formal inspection
    - Formal verification
    - Test case generation
    - Model-checking
    - Formal proof
    - informal proof
    - Animation
    - Simulation
    - Visualisation

Quality Management Activities

- **Quality assurance**
    - The establishment of a framework of organisational procedures and standards that lead to high-quality software
- **Quality planning**
    - The selection of appropriate procedures and standards from this framework, adapted for a specific software project (cf. project planning)
- **Quality control**
    - The definition and enactment of processes that ensure the software development team have followed project quality procedures and standards

So how can we ensure good quality assurance ?

- **Product standards**
    - Document standards
        - Rules that defines how to code:
            - Naming convention
            - indentation
            - Where to place {}
    - Documentation standards
        - e.g. standard comment headers for object classes
    - Coding standards
        - e.g. how some programming language is to be used
- **Process standards**
    - Definition of specification, design and validation processes
        - Fx Validation process:
            - People must have running autotest on build server, and they must be successful, in order to create pull request to code repo.
        - Definition of done:
            - Define “definition of done” rules - When is a feature done ?
                - Auto test must be present and running
                - Reviews of code must be approved
                - documented code.
    - Description of documents that should be written in the course of these processes

- Why software standards are useful

    - record best practices
    - Help to avoid repeating past mistakes
    - Provide basis for quality assurance process
    - Assist in continuity and conformity among staff
    - Reduce learning effort when starting new work

- There exists a standard for quality control called the ISO 9000 standard.
    - There exists many branches
    - A company can be reviewed as to how good there processes are in contrast to the ISO standard
    - If certified, shows the company having a good quality process

- Software quality attributes
    - Safety
    - Security
    - Reliability
    - Resilience
    - Robustness
    - Understandability
    - Testability
    - Adaptability
    - Modularity
    - Complexity
    - Portability
    - Usability
    - Reusability
    - Efficiency
    - Learability

We've been looking at Quality assurance, now we look Quality control

- Approaches to quality control
    - Reviews and inspections
    - Automated software assessment

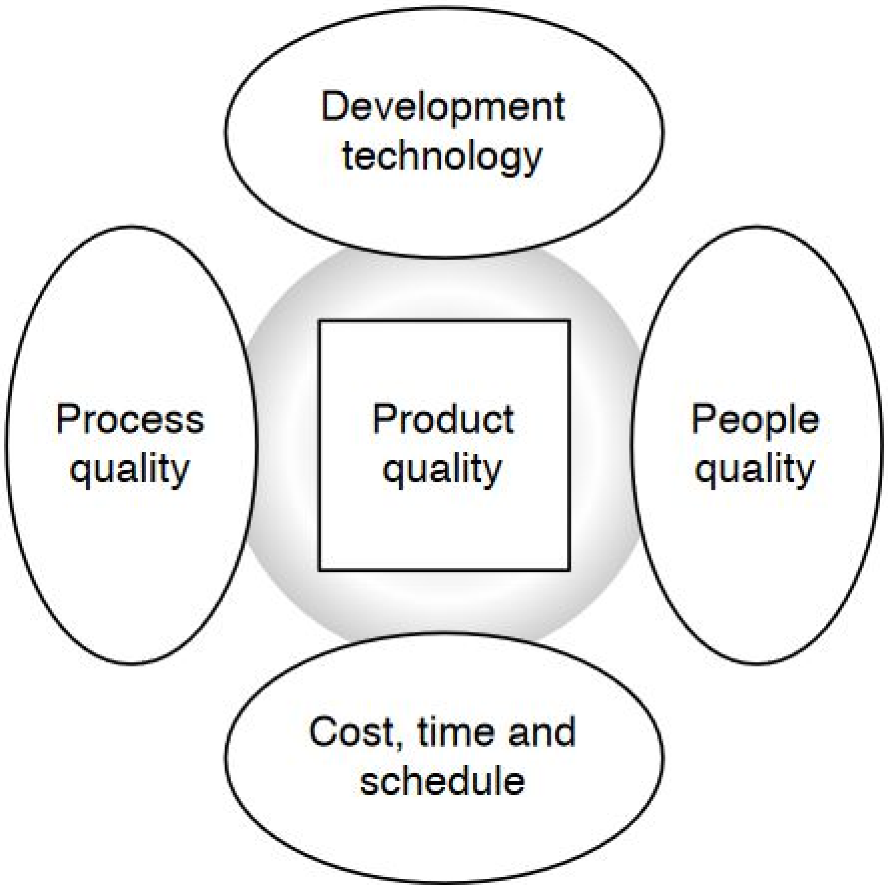
------------------- New section --------------------

- Purpose of process improvement
    - Quality of development processes and quality of developed products are closely related
    - Process improvement means:
        - Understanding existing processes and
        - Changing these process to.
            - Increase product quality
            - Reduce cost and development time

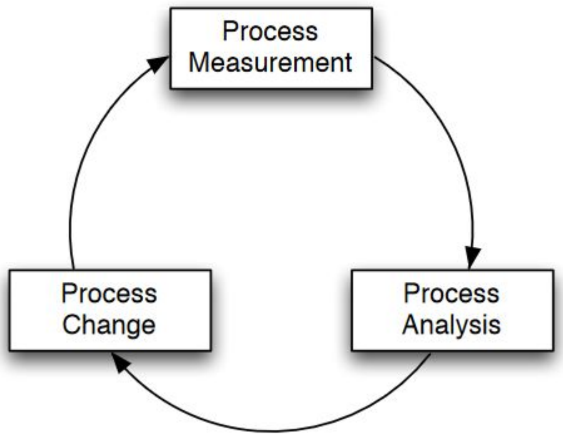
- For every process, there exists some characteristics. When trying to improve a process, we can look at the following:

Understandability	To what extent is the process defined and how easy is it to understand the process definition?
Visibility	Do the process activities culminate in clear results so that the progress of the process is externally visible?
Supportability	To what extent can CASE tools be used to line support the process activities?
Acceptability	Is the defined process acceptable to and usable by the engineers producing the software product?
Reliability	Are process errors avoided or trapped before they result in product errors?
Robustness	Can the process continue in spite of unexpected problems?
Maintainability	Can the process evolve to reflect changing organisational requirements or identified process improvements?
Rapidity	How fast can the process of delivering a system from a given specification be completed?

- Principal software Product quality factors



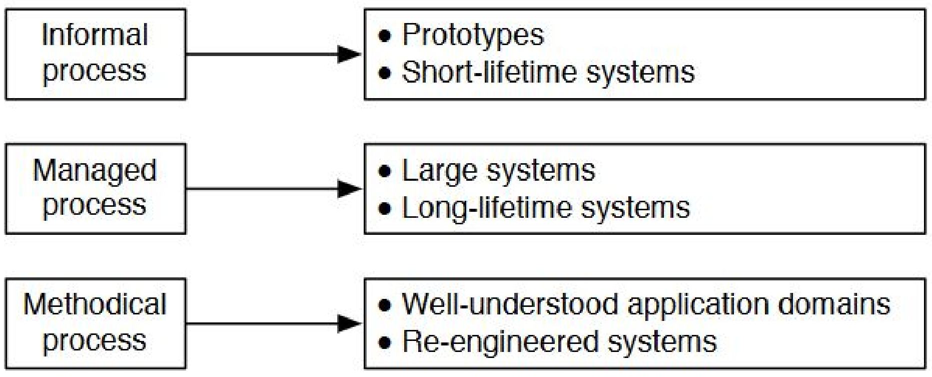
- Process improvement as cyclic activity
    - #Retrospective



- ► Process Measurement:
  measure attributes of the current process
- ► Process Analysis:
  assess current process, identify weakness and bottlenecks
- ► Process Change:
  introduce corresponding changes to the process

- Classification of processes
    - **Informal:** development team chooses the process they will use
    - **Managed:** Defined process model drives the development process
    - **Methodical:** Defined development methods supported by CASE tools (ex. scrum)
    - **Improving:** improvements considered and introduction procedures

Applicability of processes depending on classification:

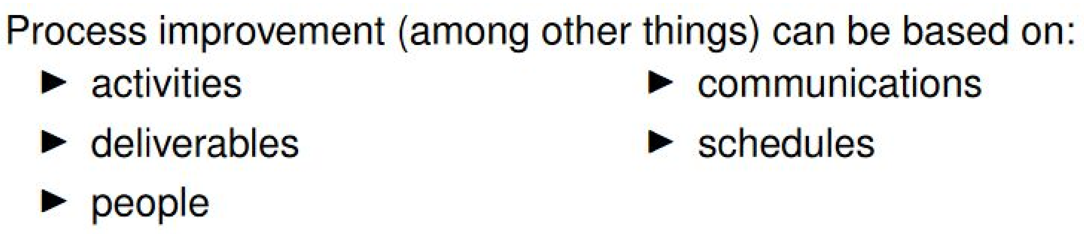


- Process measurement (how good are our processes)?
    - Time taken for a particular process to be completed
    - Resources required for a particular process
    - The number of requested requirement changes

    - Average number of lines of code modified in response to a requirement change

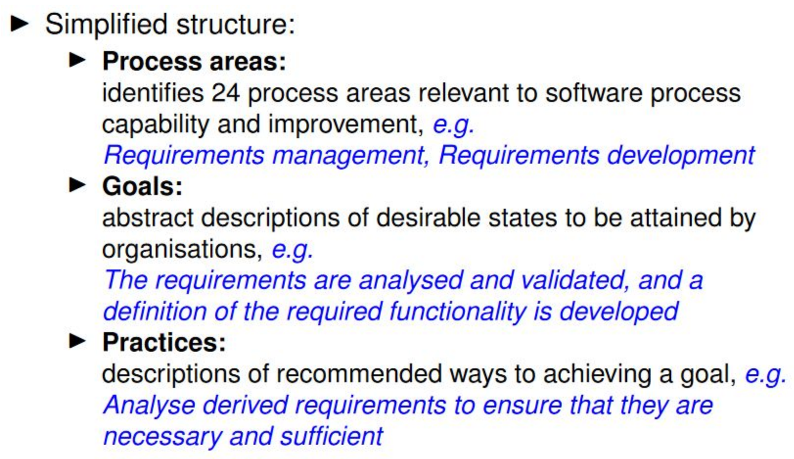
- A possible approach to measure is *Goal-Question-Metric* (GQM) paradigm
    - **Goals** what is the organisation trying to achieve?
        - e.g. increased product reliability
    - **Questions** identify specific areas of uncertainty related to goals
        - e.g. how can more effective reliability assessments be made?
    - **Metrics** Measurements to answer questions and confirm improvements
        - e.g. the number of tests required to cause product failure

- Process Analysis
    - Rely on:
        - “Formal” process models
            - Specifies activities, deliverables
        - Questionnaires and interviews
            - Question the engineers about what actually goes on
            - Refine answers by subsequent interviews
        - Ethnographic studies
            - Observer the working environment to gain understand
            - Who you talk to matters. Some angle may be adjusted

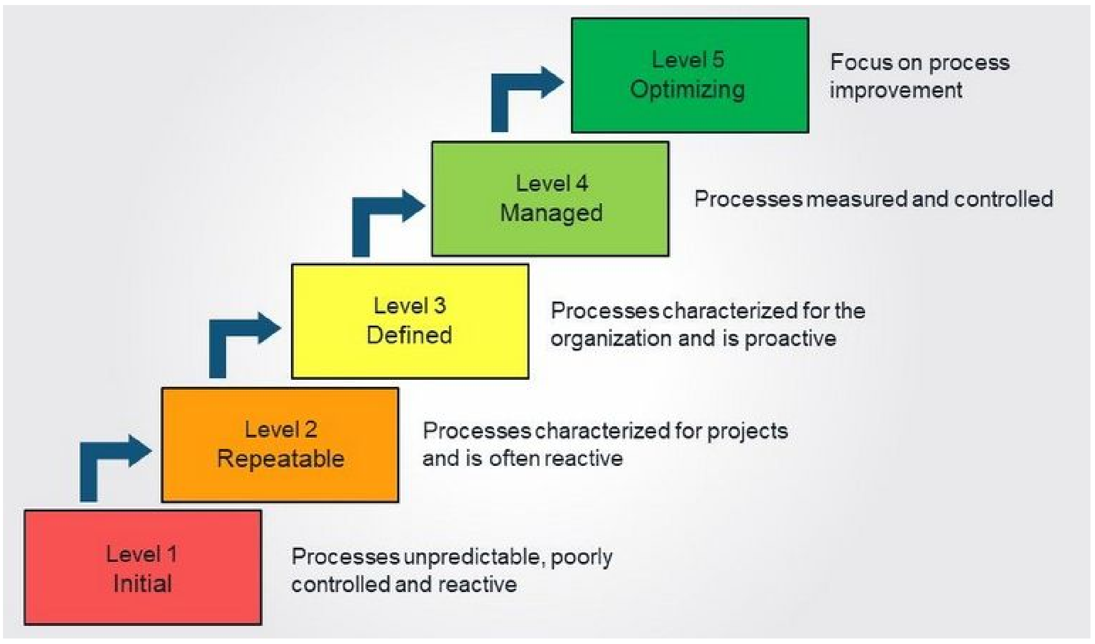

Process improvement (among other things) can be based on:
- activities
- deliverables
- people
- communications
- schedules

- Changing process phases:
    a. 1: Identify improvements
    b. 2: Prioritize the most important ones (Dont change many things at the same time)
    c. 3: Introduce process change -> train engineers to fit the change
    d. 4: get feedback from improvement changes
    e. 5: retune process changes from feedback.

- CMMI process
    - CMMI: Integrated capability maturity model
    - very complex



- Simplified structure:
    - **Process areas:**
      identifies 24 process areas relevant to software process capability and improvement, *e.g.*
      Requirements management, Requirements development
    - **Goals:**
      abstract descriptions of desirable states to be attained by organisations, *e.g.*
      The requirements are analysed and validated, and a definition of the required functionality is developed
    - **Practices:**
      descriptions of recommended ways to achieving a goal, *e.g.*
      Analyse derived requirements to ensure that they are necessary and sufficient


Level 5
Optimizing
Focus on process improvement
Level 4
Managed
Processes measured and controlled
Level 3
Defined
Processes characterized for the organization and is proactive
Level 2
Repeatable
Processes characterized for projects and is often reactive
Level 1
Initial
Processes unpredictable, poorly controlled and reactive

- FLET IND DESIGN PATTERNS ET ELLER ANDET STED
    - Coupling
    - patterns
    - cohesion
    - Principles
    - Reuse existing designs
    - layering (database layer, bla. bla.)
    - Design patterns: common problems fixed efficiently

Some Design Principles

- Divide and conquer
  (small problems are better than big problems)
- Increase cohesion
  (keep related things together)
- Reduce coupling
  (especially avoid unwanted side effects)
- Abstraction
  (information hiding)
- Reuse existing designs
  (do not copy!)
- Ensure testability
  (specify!)
- Defensive design
  (design by contract)

Concluding Remarks

How to design a good architecture?

- Follow *design principles*
- Design patterns capture *experience* and good practice
- Architectural styles are *large* design patterns
- Use design and architectural *patterns* and *styles* when possible

But,

- There is not a pattern/style for *every* problem
- Do not *overuse* patterns/styles (similarly to inheritance)
- Study patterns/styles *before* applying them
- Wrong usage will have *adverse* effect on quality indicators
- Use patterns/styles *with moderation*

5. Software Architecture

- What is Architectural Design
    - Understanding how system should be organised
    - Designing
        - Overall structure
        - Principle components
        - Relationships
        - How distributed

- What is Architectural Pattern?
    - Smart ways to structure components to solve an overall problem.
        - Examples:
            - Microservice - scalability
            - Public-subscribe structured services - real time communication
            - Client server - contact when needed
            - pipes and filters - for processing (google search - distributes MAP REDUCE)
    - Is a general, reusable solution to a commonly occurring problem in software architecture within a given context.
    - They're similar to software design patterns but have a broader scope

- Architectural Design vs Architectural Patterns
    - Architecture addresses non-functional requirements
    - Architectural design addresses structure and behaviour
- Components address functional requirements

- Architecture design - **viewpoints**
    - Architecture design are like blueprints - "look" at the system from a particular perspective
    - Cant represent all aspects of a system in one diagram
    - Different stakeholders need different info
- Krutchen 4+1
    - A way to describe the architectural representation from different aspects
    - Allows different stakeholders/Audiens to easily find what the need, and in the level of detail needed.
        - **Logical view**
            - The audience for this view, will be developer and testers, which will interested the different system layers, including any sub systems and their relations
        - **Development view**
            - The audience for this view will be developers, and specific tester that are more code involved. The view describes the different modules which the system is build of, typically drawn by UML and how the different modules relate to each other
        - **Procces view**

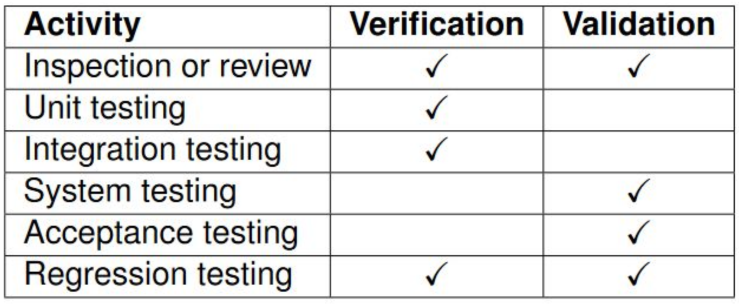
    - The audience for this view will be integrators and programmers. the area describes non-functional requirements, the design’s concurrency and synchronization aspects and elaborates the runtime behaviour of the system
  - **Physical view**
    - The audience for this view will be System engineers and Deployment managers. The physical view area describes the architecturally significant persistent elements in the data mode as well as the topology. Describes the mapping of the software into hardware and shows the system’s distributed aspects

6. Validation and Verification

- Introduction
- Testing Basics
- Test Case Design
- Testing methods
- White Box Testing
  - Unit Testing
  - Integration Testing
- Black Box Testing
  - System testing
  - Acceptance
  - Regression Testing
- Concluding Remarks

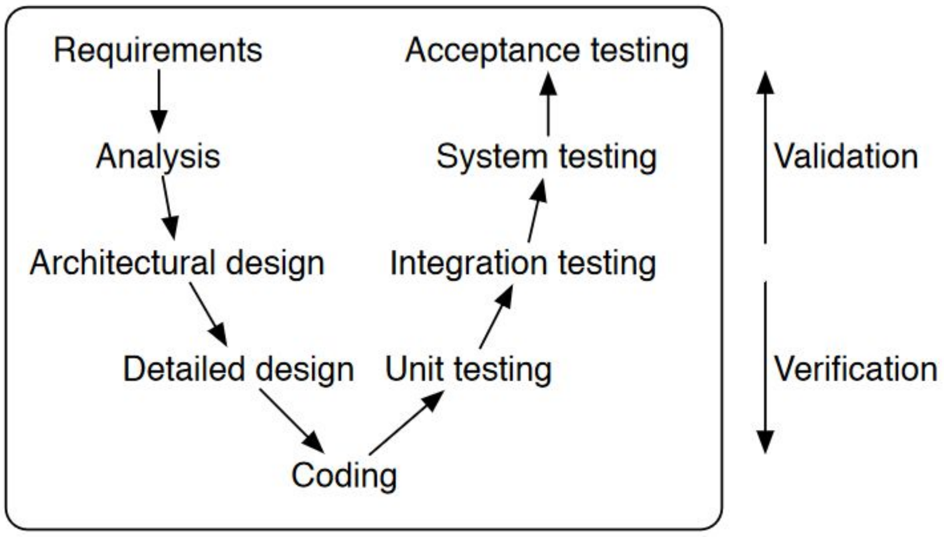
Which problems are we trying to solve?

- Are we building the right product?
  - Validation
- Are we building the product right?
  - Verification
- Does the product satisfy the specification?



Activity	Verification	Validation
Inspection or review	✓	✓
Unit testing	✓	
Integration testing	✓	
System testing		✓
Acceptance testing		✓
Regression testing	✓	✓

- Verification and validation testing in the v-model



What are the objectives of testing?
- Find errors
- Check for compliance with requirements
- Break the software?
- Reduce risk (you cant)
    - Cant test for everything
- Check performance
- Show absence of defects? (you cant)
    - "Program testing can be used to show the presence of defects, but never their absence"

Program complexity causes problems
- Estimate time to test addition of two 32 bit integers thoroughly!
- 2^64 values to inspect
- How many seconds, minutes, years, centuries, millennia?
- A reasonable estimate would be approx. 600,000 millenia
    - if we run 1000 tests per second

Ad-Hoc testing
- Belongs to the **code & fix** development "method" (least formal test method)
    - You have implemented something
    - You run it
    - You look whether you are satisfied with the result
    - if not, you fix the program

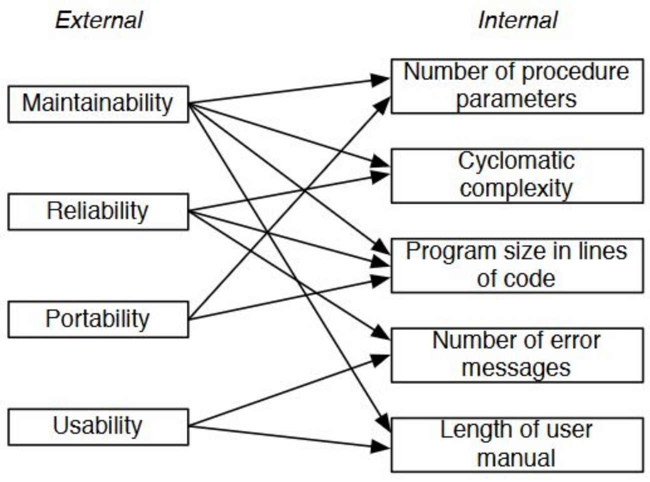
Analogy: Scientific experiment
- To find out whether some process works
- Need to state the expected before the experiment
- Must know the precise conditions under which the experiment runs
- the experiment must be repeatable

Software measurement and metrics

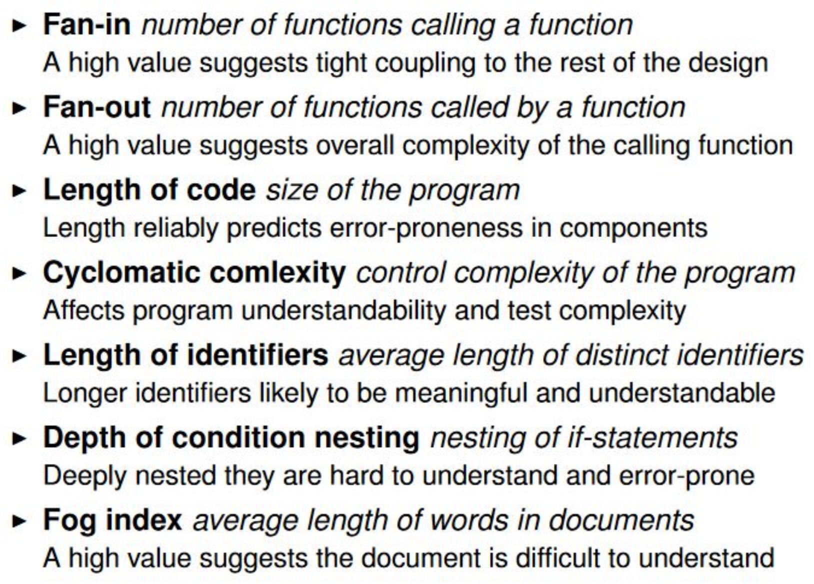
- Aims
  - Make general predictions about a system
  - identify anomalous components
- Classes of metrics
  - Dynamic: measured during program execution (e.g. execution times of specific functions)
    - Whats takes RAM
    - Whats takes a lot a time
  - Static: measured by means of the software artefacts (e.g. program size in lines of code)

External and internal software attributes

- Internal attributes must be measured accurately
- A well-understood relationship between the attributes must exist



Static Software Product Metrics
Fan-in: Coupling
Fan-out: Coheison



- **Fan-in** *number of functions calling a function*
  A high value suggests tight coupling to the rest of the design
- **Fan-out** *number of functions called by a function*
  A high value suggests overall complexity of the calling function
- **Length of code** *size of the program*
  Length reliably predicts error-proneness in components
- **Cyclomatic comlexity** *control complexity of the program*
  Affects program understandability and test complexity
- **Length of identifiers** *average length of distinct identifiers*
  Longer identifiers likely to be meaningful and understandable
- **Depth of condition nesting** *nesting of if-statements*
  Deeply nested they are hard to understand and error-prone
- **Fog index** *average length of words in documents*
  A high value suggests the document is difficult to understand

Planning Software tests

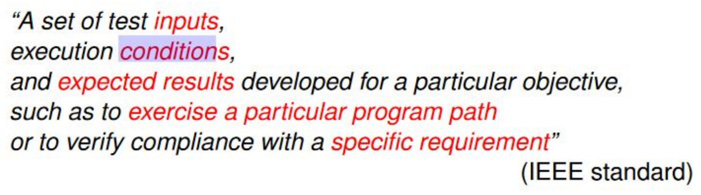
- Device a test plan:
    - Testing process
    - Requirements traceability
    - Test items
    - Testing schedule
    - Test recording procedures

What should a test measure?

- Achieve an acceptable **level of confidence** (fx. boundary analysis) that the system **behaves correctly** (when given specific input, i expect this…) under all **circumstances of interest**

What is a test case?

- fx. Arrange, Act, Assert (AAA. pattern)



"A set of test inputs,
execution conditions,
and expected results developed for a particular objective,
such as to exercise a particular program path
or to verify compliance with a specific requirement"

(IEEE standard)

Testing methods

- White box testing
    - Exercises program: conditions, loops, data structures
        - unit test (take different path for 100% coverage)
        - integration test. Different methods
            - Big Bang, Bottom-up, Top-down, Collaboration, Sandwich
- Black box testing (typically used in later testing stages)
    - Ignores implementation details
    - Exercises all functional requirements
        - Boundary Value Analysis and Equivalence Partitioning
        - System tests
        - Acceptance tests

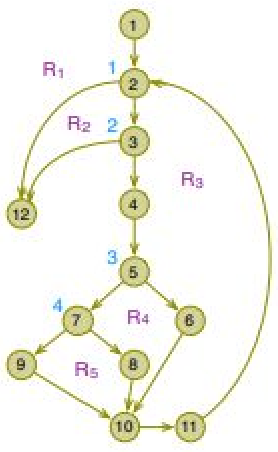
Worked example: Binary search

Cyclomatic complexity V(G)

$$= 15 - 12 + 2$$
$$\#(\text{edges}) - \#(\text{nodes}) + 2$$

$$= 4 + 1$$
$$\#(\text{predicate nodes}) + 1$$

$$= 5$$
$$\#(\text{regions})$$



basis paths?

- System testing
- Acceptance testing
- Regression testing
  - If possible, automate

Concluding Remarks

- Testing must be *systematic*
- Testing begins with *requirements*
- Testing must be taken into account *during design*
- *White box testing* looks inside the program
- *Unit tests* tests the smallest units of a system
- *Integration tests* test communication between components
- *Black box testing* ignores the inside of the program
- Testing does *not* address all correctness problems
- Need for *complementary techniques*
  - Inspection
  - Formal methods

What is Combinatorial testing (ikke helt styr på det)

- For systems with alot of different outcomes, combinatorial testing is used
    - fx. a system with on-off switches:

      · 34 switches = 2^{34} = 1.7 x 10^{10} possible inputs = 1.7 x 10^{10} tests

    - Instead of an exhaustive search of all combinations of all parameters, combinatorial testing can allow for optimized searching via carefully chosen test vectors...testing all discrete combinations of those parameters

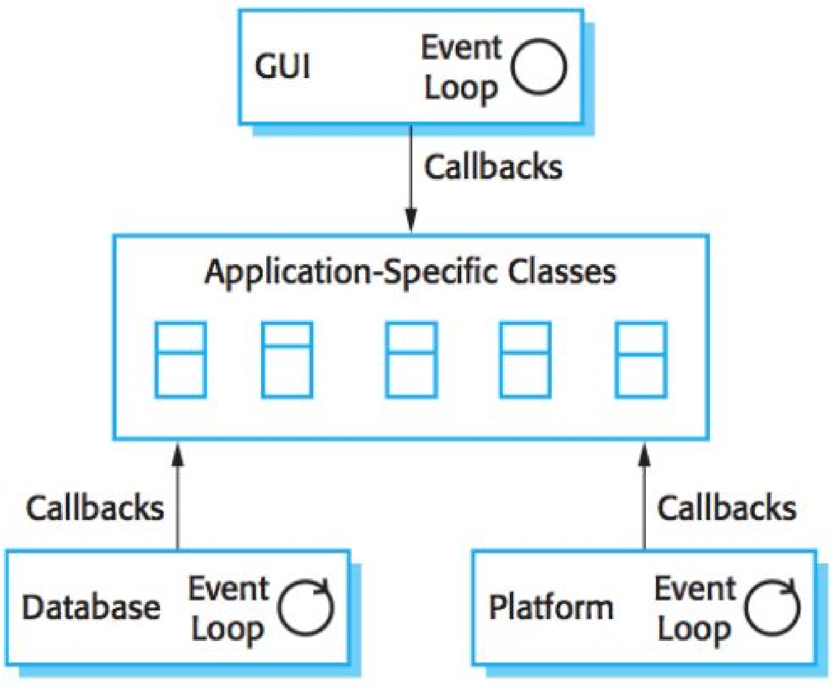
Validation using simulation

- VDM model can be used for improve the requirements specification
- Starting with simulations of the system might reveal problems at an early start, rather that at the end of the project.
- Starting with simulations early, makes on scope of each req easier to time estimate reasons that reveal problems can be included in the estimate.

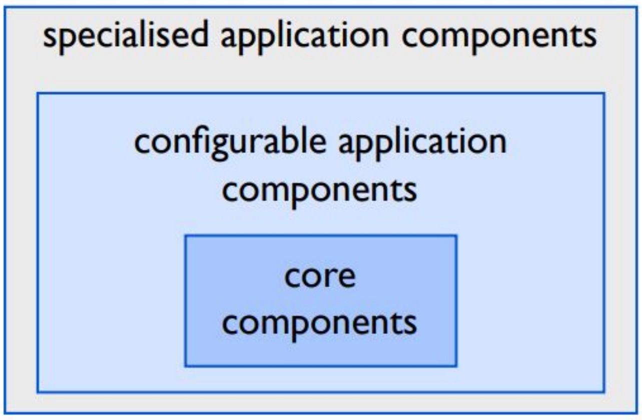
7. Reuse

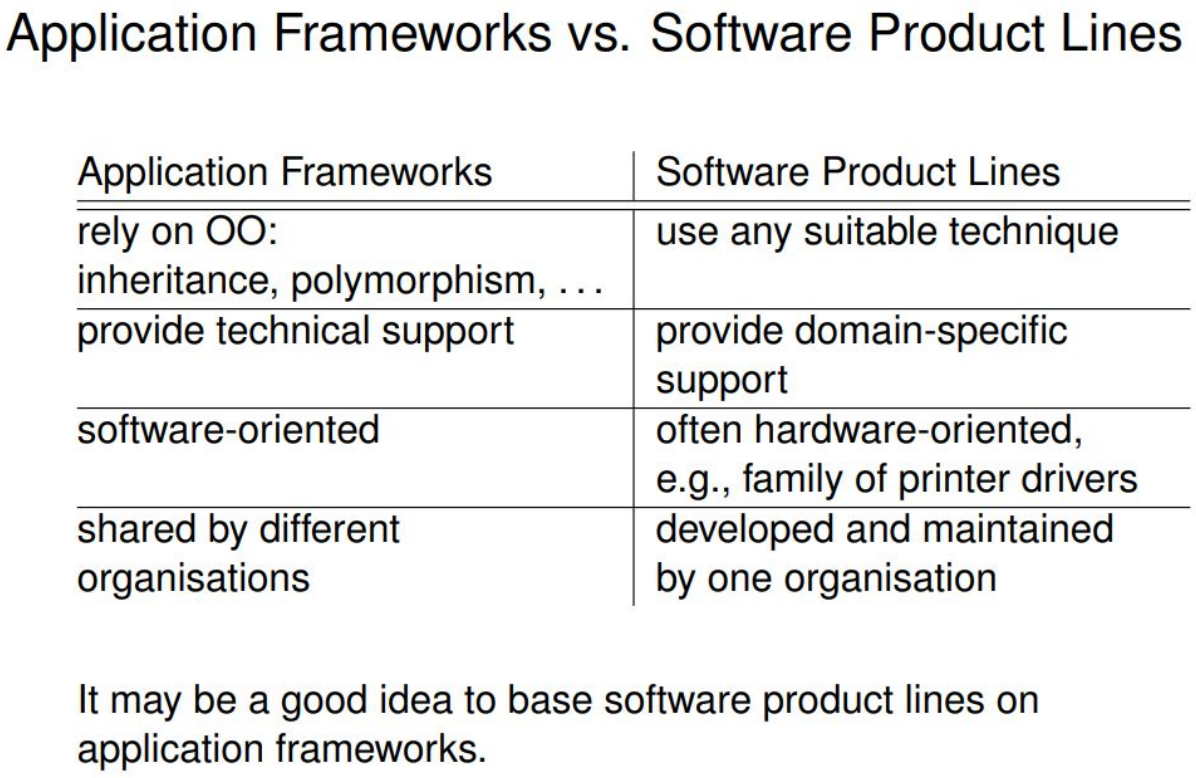
- Reuse
    - Why Reuse?
    - Reuse-base software engineering- reuse existing software as much as possible
    - Different scales
        - System reuse
            - number of applications incorporated into larger "system of systems"
        - application reuse
            - existing application incorporated into other system (e.g. software product lines)
        - component reuse
            - entire subsystems or just small modules
        - object function reuse
            - single functions, standard libraries offer this
            - in practice re-using specific components tricky
        - concept reuse
            - e.g. "design patterns" (coming in a later lecture...)
    - What were the early problems with reuse?
        - The early hope of OO programming was to reuse object classes
            - But in practice objects too fine-grained to be reused and it was quicker just to re-implement it.

- Then came the idea to reuse Application framework. An idea to reuse software at a higher level of abstraction
    - Not fine-grained object but larger-grained “framework”
    - Application framework: provides a generic structure of an application
        - You “extend” (in sense of OO) to implement your functionality
        - It is software - a collection of abstract and concrete classes (if it’s an OO framework)
        - An example of an application framework could be “bare-bones” MVC software. You extend certain classes to implement particular model
        - Framework definition: “Dont call us, we call you”. For WPF, we define ui elements, and business logic for the views. The framework WPF, calls our code, and renders it.



- Software Product Line
    - When company needs to support many similar (but not identical) systems
    - e.g Printer manufacturer
        - Every printer has its own control software
        - Control software very similar
        - So make core product - adapt for each printer
        - Common architecture + shared components + some specialisation

        - 
specialised application components
configurable application components
core components

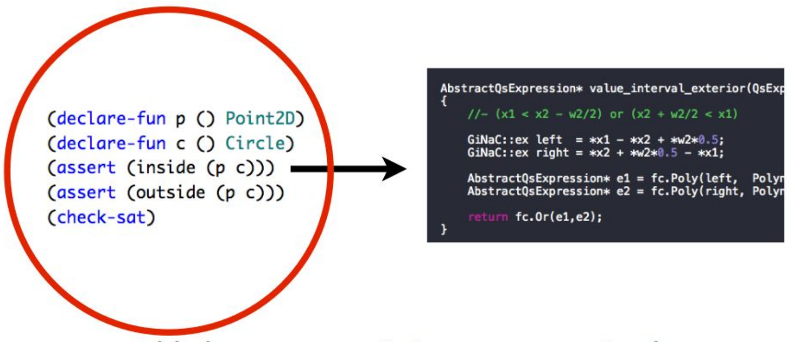
- But what happens if the software “evolves”? What does this mean for core components?
    - 

Application Frameworks vs. Software Product Lines

Application Frameworks	Software Product Lines
rely on OO: inheritance, polymorphism, . . .	use any suitable technique
provide technical support	provide domain-specific support
software-oriented	often hardware-oriented, e.g., family of printer drivers
shared by different organisations	developed and maintained by one organisation

It may be a good idea to base software product lines on application frameworks.

- How to configure or adapt a software product line
    - One possibility use a domain-specific language (DSL)
    - Encode the domain knowledge in the DSL
    - Implement functionality of the product in the DSL
        - tailored to a big domain
            - Language made for only a specific domain
            - NOT a general purpose language like C#, c++ etc.
        - Examples:
            - SQL
            - HTLM



```
(declare-fun p () Point2D)
(declare-fun c () Circle)
(assert (inside (p c)))
(assert (outside (p c)))
(check-sat)
```

```
AbstractQsExpression* value_interval_exterior(QsExp
{
    //- (x1 < x2 - w2/2) or (x2 + w2/2 < x1)

    GiNaC::ex left  = *x1 - *x2 + *w2*0.5;
    GiNaC::ex right = *x2 + *w2*0.5 - *x1;

    AbstractQsExpression* e1 = fc.Poly(left,  Polyn
    AbstractQsExpression* e2 = fc.Poly(right, Polyn

    return fc.Or(e1,e2);
}
```

we could also use an existing programming language
(e.g. this example is in SMT-LIB language)

8. Safety Critical Software

Safety Critical Systems: Failure leads to loss of
- life
- health
- well-being

Mission Critical Systems: Failure leads to loss of
- Business
- Equipment
- Effor

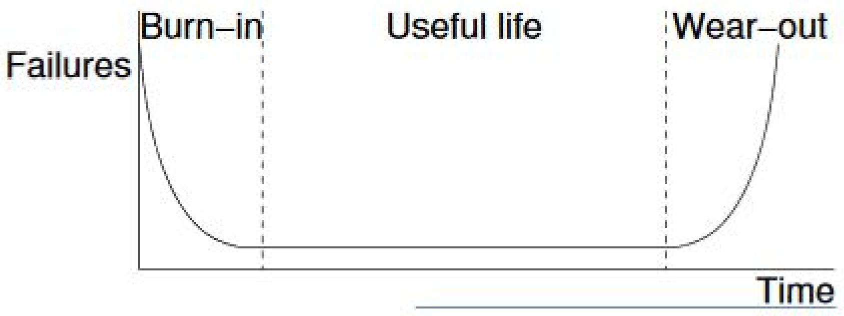
Security critical systems: Failure leads to loss of
- Confidentiality
- Integrity
- Availability of service

~~Failsafe systems~~
- ~~**Failsafe system:** a system designed to fail in a safe state~~
- ~~**Failure:** the event of a system or component failing to perform, or deviating from its intended/required function for a specified time under specified environment conditions~~
- ~~**Error:** deviation from an intended/required, correct state of the system or subsystem~~
- ~~**Fault:** the real of hypothesised cause of an error (actual or potential)~~

~~Primary and secondary faults~~
- ~~**Primary fault:** a system or component fails **within** the intended operating environment or conditions~~
- ~~**Secondary fault:** a system or component fails **outside** its intended operating environment or conditions~~

~~Useful lifetime~~

- ~~**Early failures:** occur during a **debugging** or **burn-in period**~~
- ~~**Random failures:** result from complex, uncontrollable causes usually primarily considered **during the useful life** of the component or system;~~
- ~~**Wearout failures:** occur when the components or systems are **past their useful life**~~



Safety-critical system: therac-25

- Radiation therapy machine (between 1985 and 1987, 6 people wrongly received **massive overdoses of radiation**

Lessons learned Process

- Overconfidence in software
- Confusing reliability with safety
- Lack of defensive
- Failure to eliminate root causes
- Complacency
- Unrealistic risk assessments
- Inadequate investigation or follow-up on accident reports
- Dangerous reuse of software
- Safe versus friendly user interface
- Importance of government regulator (FDA)

~~Lessons learned engineering practices~~

- ~~Inadequate software engineering practices:~~
    - ~~Software specification and documentation were an afterthought~~
    - ~~Bad quality assurance procedures~~
    - ~~Software design was unnecessarily complex~~
    - ~~No audit trails~~
    - ~~Inadequate testing and formal analysis of code~~
    - ~~Bad user interface design, in particular, incomprehensible error message~~

Safety in system development

- Safety should be treated as a **separate** (though related) issue to other system requirements during development
- Important to have a **safety plan**

- Safety plan
    - Set out the manner in which **safety** will be **achieved**
    - define management structure responsible for **hazard** and **risk** analysis
    - Identify key **staff**
    - Assign **responsibilities** within project performed by large companies or consortia

Software standards

- Help to ensure that a product meets certain levels of **quality**
- Help to establish that a product has been developed using methods of **known effectiveness**
- Promote **uniformity of approach** between different teams
- Provide **guidance** on design and development techniques
- Provide **legal basis** in case of dispute

Programming language for critical systems

- Desirable language feature include: An example could be **ADA**
    - Logical soundness and clear precise definition
    - Expressive power
    - Strong typing
    - Verifiability
    - Security (language violations can be checked statically)
    - Bounded space requirements
    - Bounded Time requirements
    - Good tool support

Common software faults

- Subprogram side-effects
- Aliasing
- Failure to initialise
- Expression evaluation error
- Control flow errors

Language subsets

- Use of a subset of a programming language with some or all of these features:
    - No gotos
    - No pointers
    - No dynamic data structures
    - No exceptions
    - No tasking
    - No side-effects

    - No generics
- Java is therefore a no-go

Static analysis
- Investing properties of a system **without operating it:**
    - Type checking
    - Formal verification/refinement
    - Control flow analysis
    - Walkthroughs / design reviews
    - Metrics complexity measures (Cyclomatic / McCabe etc)

Dynamic analysis
- Functional testing **Black Box**
    - Use test cases to check that the system has the required functionality
    - No knowledge of implementation assumed
    - Test cases generated from formal specification
- Structural Testing **White Box**
    - Investigate characteristics of systems using detailed knowledge of implementation
    - Device test cases to check individual subprograms/execution paths
- Random testing
    - Random selection of test cases
    - Aims to detect faults that may be missed by more systematic techniques
- Test coverage
    - Statement, branch(decision), MCDC, call graph

Specific Testing techniques
- Error seeding - edit the program code to have an error, and make sure that the written test catches the error - Its like reverseve testing
- Boundary value analysis (test software at, and at either side of, each boundary value)
- Error guessing of possible faults based on experience
- Performance testing
- Stress testing

ENVIRONMENTAL SIMULATION
- For control/protection systems, it is impossible to fully test a system within its operational environment
- Simulator of the environment used instead
- Often more complex than system under test!
- Issues:
    - Which environmental variables are included
    - Accuracy of models
    - Accuracy calculations
    - Timing considerations

~~**Maintenance**~~
- ~~**Maintenance:** the action taken to retain a system in, or return a system to, its design operation condition~~
- ~~**Maintainability:** the ability of a system to be maintained~~

~~**Safety**~~

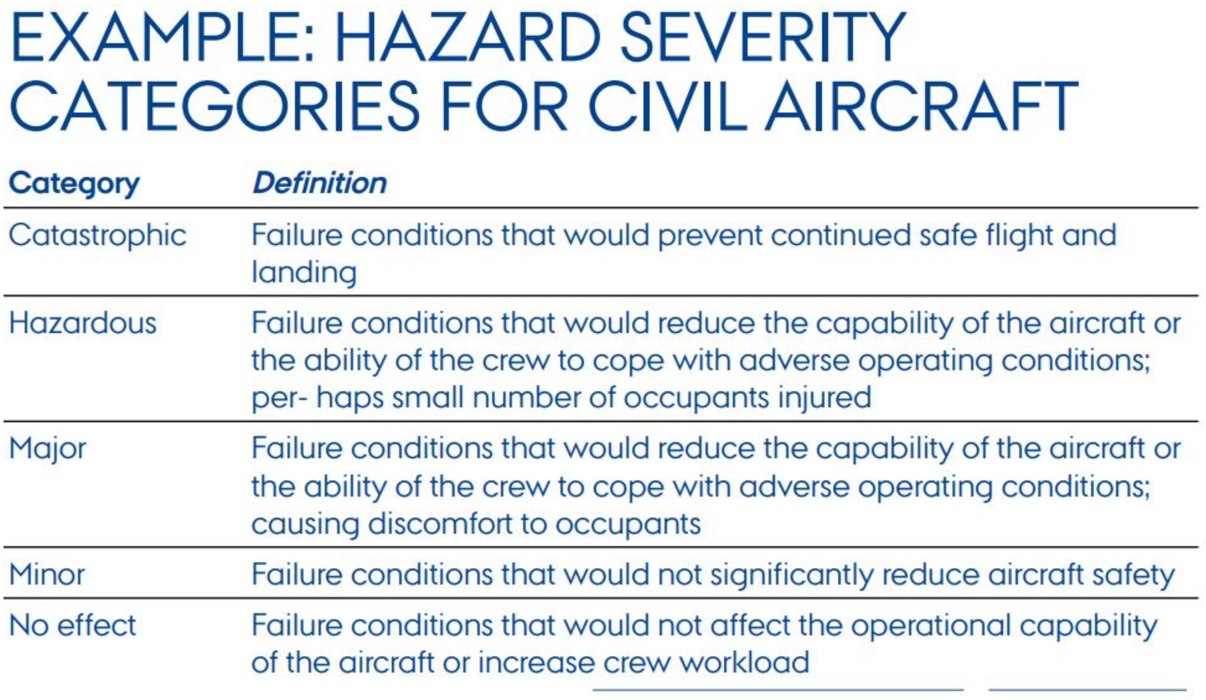
- ~~**Accident:**~~
  - ~~unintended event or sequence of events that causes death, injury, environmental or material damage~~
  - ~~unplanned event that results in a certain level of damage or loss to human life or the environment~~
- ~~**Incident :**~~
  - ~~unplanned event that involves no loss or damage, but has the potential to be an accident in different circumstance~~
- ~~**Safety:**~~
  - ~~Freedom of a system from accidents or losses~~

~~**RELIABILITY**~~

- ~~**Reliability:** The probability that a system or component will perform its intended function satisfactorily for a prescribed period of time under a given set of operating conditions~~
- ~~**Safety and reliability:** S system can be unreliable but safe if it does not behave according to its specification but still does not cause an accident~~

HAZARD

- **Hazard:** A situation in which there is the danger of an accident occurring
- **Hazard severity:** The worst possible accident that could result from the hazard
- **Hazard likelihood/probability:** can be specified qualitatively or quantitatively when a new system is designed usually no historical data is available so qualitative evaluation may be the best that can be done



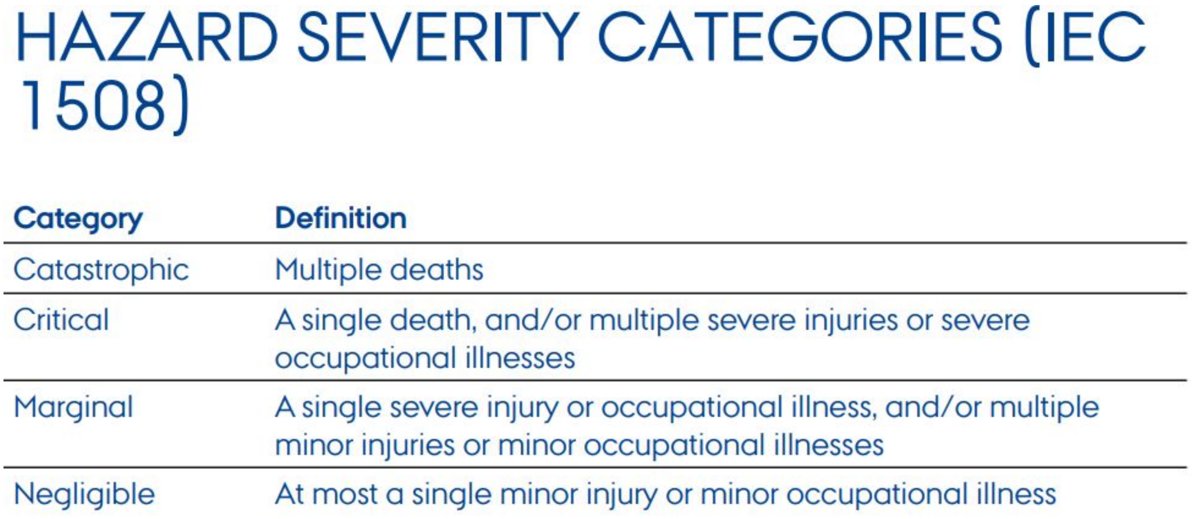
EXAMPLE: HAZARD SEVERITY CATEGORIES FOR CIVIL AIRCRAFT

Category	***Definition***
Catastrophic	Failure conditions that would prevent continued safe flight and landing
Hazardous	Failure conditions that would reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions; per- haps small number of occupants injured
Major	Failure conditions that would reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions; causing discomfort to occupants
Minor	Failure conditions that would not significantly reduce aircraft safety
No effect	Failure conditions that would not affect the operational capability of the aircraft or increase crew workload

RISK

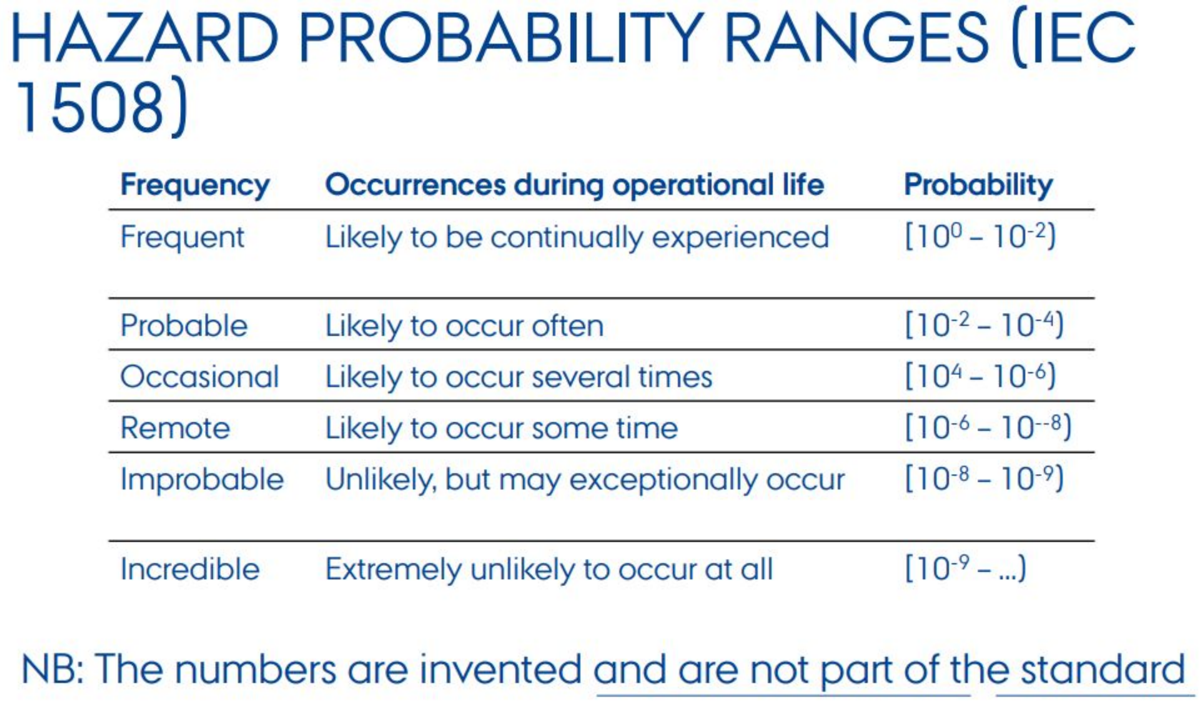
- **Risk:** combination of the **severity** of a specified hazardous event with its **probability** of occurrence over a specified duration
- **Example:** Failure of a particular component results in an explosion that could kill 100 people. It is estimated that the component will fail once every 10000 years. ›What is the

risk associated with this component? ›Risk = severity × probability of occurrence per year = 100 × (1/10000) = 0.01 deaths per year



HAZARD SEVERITY CATEGORIES (IEC 1508)

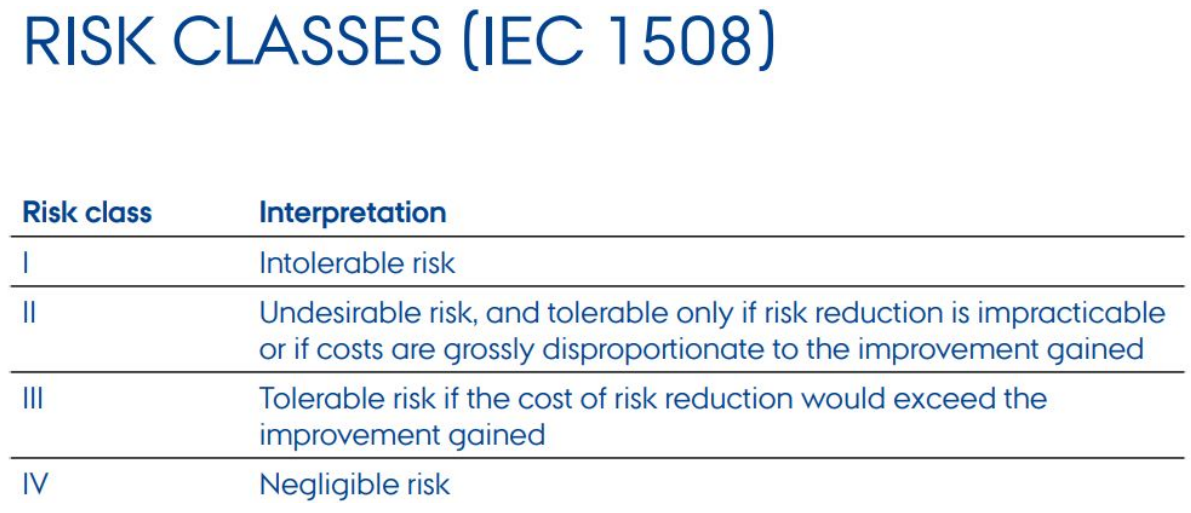
Category	Definition
Catastrophic	Multiple deaths
Critical	A single death, and/or multiple severe injuries or severe occupational illnesses
Marginal	A single severe injury or occupational illness, and/or multiple minor injuries or minor occupational illnesses
Negligible	At most a single minor injury or minor occupational illness



HAZARD PROBABILITY RANGES (IEC 1508)

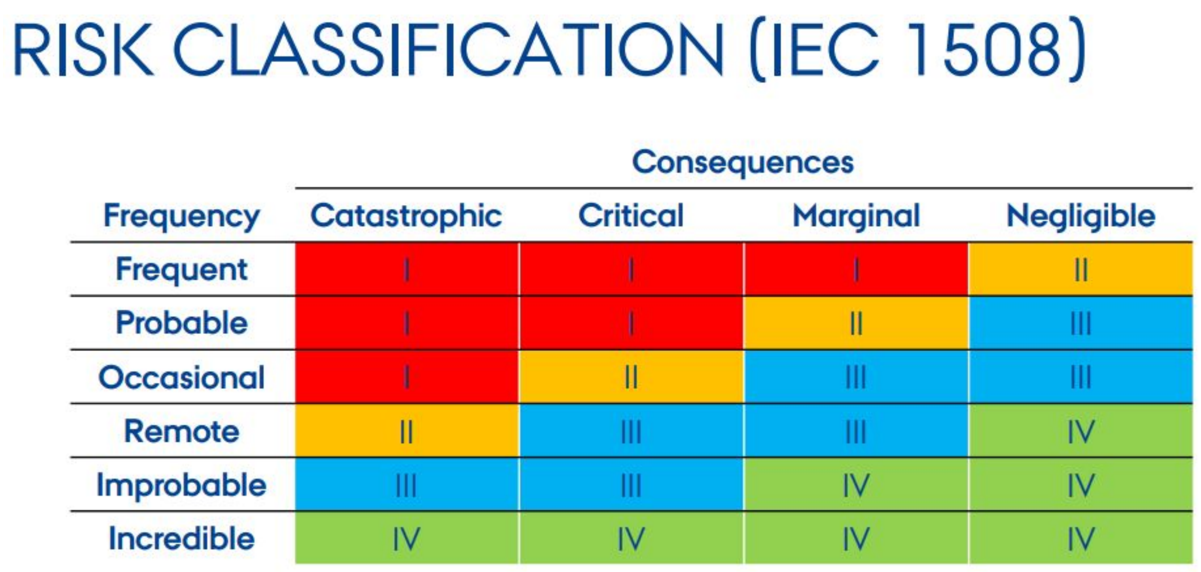
Frequency	Occurrences during operational life	Probability
Frequent	Likely to be continually experienced	$[10^{0} – 10^{-2})$
Probable	Likely to occur often	$[10^{-2} – 10^{-4})$
Occasional	Likely to occur several times	$[10^{4} – 10^{-6})$
Remote	Likely to occur some time	$[10^{-6} – 10^{--8})$
Improbable	Unlikely, but may exceptionally occur	$[10^{-8} – 10^{-9})$
Incredible	Extremely unlikely to occur at all	$[10^{-9}$ – ...)

NB: The numbers are invented and are not part of the standard



RISK CLASSES (IEC 1508)

Risk class	Interpretation
I	Intolerable risk
II	Undesirable risk, and tolerable only if risk reduction is impracticable or if costs are grossly disproportionate to the improvement gained
III	Tolerable risk if the cost of risk reduction would exceed the improvement gained
IV	Negligible risk



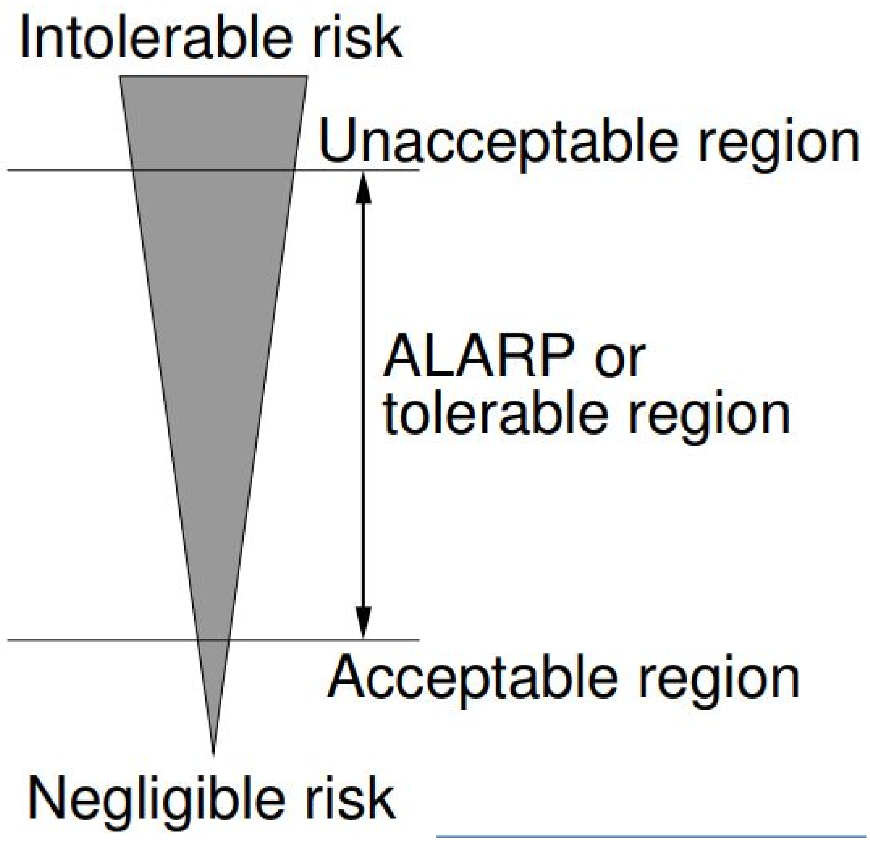
RISK CLASSIFICATION (IEC 1508)

	Consequences			
Frequency	Catastrophic	Critical	Marginal	Negligible
Frequent	I	I	I	II
Probable	I	I	II	III
Occasional	I	II	III	III
Remote	II	III	III	IV
Improbable	III	III	IV	IV
Incredible	IV	IV	IV	IV

ACCEPTABILITY OF RISK

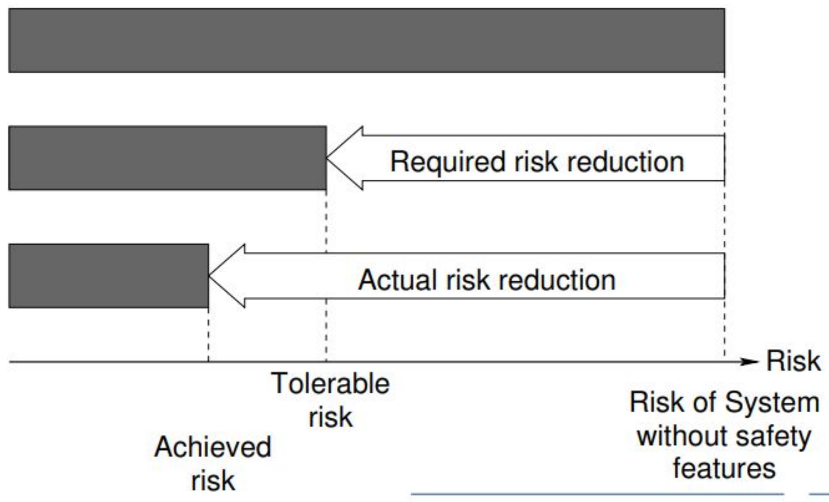
- **ALARP: As Low As is Reasonably Possible**
- If risk can easily be reduced, **it should be Conversely**, a system with significant risk may be acceptable if it offers sufficient benefit and if further reduction of risk is impracticable or incurs unjustifiable cost
- Risk level satisfying ALARP is termed **tolerable risk**

ALARP



RISK MANAGEMENT AND REDUCTION

Producing a safety-critical system can be seen as a process of risk reduction:

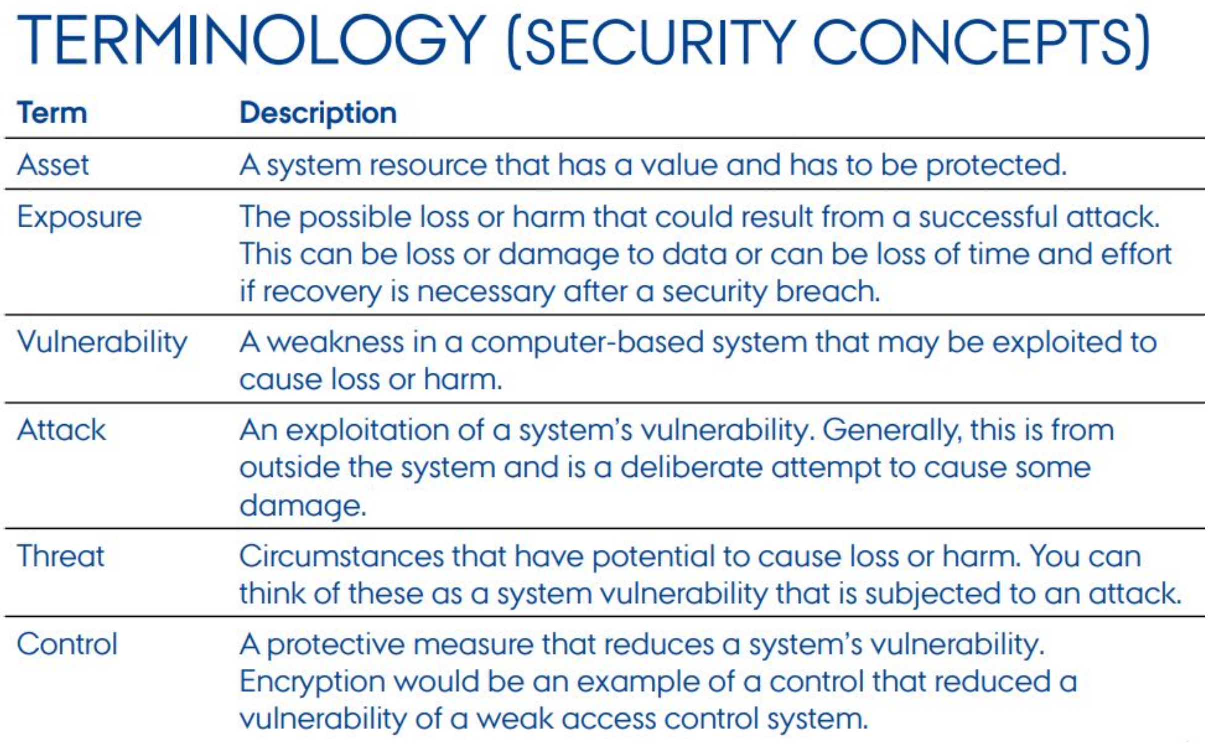


ETHICAL CONSIDERATIONS

- Determining risk and more especially acceptability of risk involves morals/judgements/opinions
- Society’s view is not easily determined by a clear set of rules - unpredictable - illogical
- Perception: **accidents involving large numbers of deaths** (e.g. Enschede train derailment— 101 deaths) are perceived as being **more serious than smaller accidents** though they may occur less frequently (e.g. German road accident fatalities — approx 250/month)
- Various professional societies offer guidelines on risk management

~~**Safe vs Secure**~~

- ~~**Safe:** The system shall … ensure safe operation~~
- ~~**Secure:** The system shall not … be compromised~~
- ~~How to develop and maintain systems that can resist malicious attacks intended to damage a computerbased system or its data?~~



TERMINOLOGY (SECURITY CONCEPTS)

Term	Description
Asset	A system resource that has a value and has to be protected.
Exposure	The possible loss or harm that could result from a successful attack. This can be loss or damage to data or can be loss of time and effort if recovery is necessary after a security breach.
Vulnerability	A weakness in a computer-based system that may be exploited to cause loss or harm.
Attack	An exploitation of a system's vulnerability. Generally, this is from outside the system and is a deliberate attempt to cause some damage.
Threat	Circumstances that have potential to cause loss or harm. You can think of these as a system vulnerability that is subjected to an attack.
Control	A protective measure that reduces a system's vulnerability. Encryption would be an example of a control that reduced a vulnerability of a weak access control system.

~~Analogies of Safety/Security~~

- ~~Compare~~
    - ~~Exposure vs Accident~~
    - ~~Vulnerability vs Hazard~~

Critical Systems

- Become a domain expert for safety, security or mission criticality
- Become a domain expert for the application domain
- What are the hazards, vulnerabilities, …. ?
- Or have such experts on your team

9. Change Management / Configuration Management

For configuration management, we talk about 4 main activities:

- Change management
    - Involves **keeping track of requests for changes** to the software from customers and developers
    - Working out the **cost and impact of making these changes** and deciding if and when the changes should be implemented
- Version management
- System building
- Release management