1. Software Development & Project Management

SKAL

Agenda:

- Firstly, what is Software Development?
    - Consists of software processes
        - What is the aim?
        - what is good sw?
            - Correctness
            - Reliability
            - Testability
            - Efficiency
- The Waterfall model
    - System Requirements > Software Requirements > Analysis > Program Design > Coding > Testing > Operations.
    - Problems ? - Testing occur late
- The V-model. Improves on the waterfall model. Emphasises on validation and verification
    - Requirements > Analysis > Architectural design > detailed design > coding > unit testing > integration testing > system testing > acceptance testing
- Agile methods
    - The agile manifesto
        - Problem: To much planning and organisation
        - Individuals and interactions over processes and tools
        - Working software over comprehensive documentation
        - responding to change over following a plan
    - Srum
        - Typically done in teams 7-8 people
        - Planning & Prioritizing
        - Sprints
        - Retrospective
        - burndown
- Comparison
    - Advantages, disadvantage, when to use

- Project Management
    - Planning, Organising, staffing, directing, monitoring, controlling, representing
    - Why is it important? (problems)
        - Poor estimate and plans, Lack of quality standards, and measures
        - Lack of techniques to make progress visible
        - Poor role definition - who does what?
        - Incorrect success criteria

- Main steps for a new project
    - Feasible Study → Plan → Execute
    - Feasible Study
        - Cost-benefit analysis
            - Creating cost & Operational costs
        - Create business model
        - Project plan must keep business case intact
    - Plan
        - Risk categories
            - Project risks,
            - product risks
            - business risks
            - Risk matrix
        - (if time)Product Breakdown structure (PBS)
        - (if time)Work breakdown structure (WBS)
        - (if time)Product flow diagram (PFD)
        - (iftime)Activity network planning
    - Execute plan
        - Visualise progress
            - Draw gantt charts
                - Risk path
                - Delays the whole projects

2. Requirements Description and Elicitation

Good

Agenda

- Why spend time on requirements?
    - The hardest single part of building a sw system is deciding precisely what to build
    - Difficult to rectify later
- What is a requirement specification?
    - Exact statement of the particular needs to be satisfied or essential characteristics that a customer requires and which vendor must deliver
- Who uses them?
    - **Managers** : for scheduling and measuring progress
    - **Customers** : for their contract
    - **Software designers**: what to design
    - **Coders**: acceptable implementations and the output that must be produced (what to build)
    - **QA personnel**: Validation, test, planning and verification

- Example of requirements (DON'T MENTION, #BACKUP)

- From **library system** (before):

RX	Every order shall be allocated a unique identifier, which the user shall be able to copy to the account's permanent area

- From **library system** (after):

R 3:1	Every book order shall be given a unique identifier

R 4:1	The user shall be able to copy to the account's permanent area

- A good requirement specification
  - **Correct** - Specifying something actually needed
  - **Unambiguous** - Only interpretation
  - **Complete** - Includes all signification
  - **Consistent** - no requirements can be verified
  - **Verifiable** - All requirements can be verified.
  - **Modifiable** - Changes can easily be made to the requirements
  - **Traceable** - The origin of each requirement is clear

- For guidelines on how to create good software requirement specification:
  - SRS IEEE
    - Functional reqs
      - Usecase vs Task description
    - Non function reqs

- Exists a model for the requirement process
  - **Tegn modellen**
    - 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

- 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.

- Why is is 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

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

Activities for getting requirements

- Interview stakeholders
  - Quantitative/Qualitative
- Brainstorming with current and potential users
- Observe current system
  - Think-out-loud

- 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

3. Requirements Validation

- **Agenda**
- **Draw diagram**
- **Validation Vs verification**
  - **What is a Requirement validation**
    - Demonstrate that the specification matches the requirements
  - Elicitation
    - Formulating overall goals of new system (Mission statement)
    - Describing the users current work process and its problems.
  - Analyse
    - DONT MENTION (Assumptions + specification implies)
    - Analyse the elitication data

- Specification
  - Specification of requirements
    - Use case
    - Task description
    - Sequence diagram
    - State diagrams
  - Requirements
    - **Correct** - Specifying something actually needed
    - **Unambiguous** - Only interpretation
    - **Complete** - Includes all signification
    - **Consistent** - no requirements can be verified
    - **Verifiable** - All requirements can be verified.
    - **Modifiable** - Changes can easily be made to the requirements
    - **Traceable** - The origin of each requirement is clear
- Validation
  - Demonstrate that the specification matches the requirements
  - Why is spec hard ?
    - What are the problems no a bad validation?
    - Why do we need a good validation before development?
      - Building the correct produkt
      - Analyse early problems that can delay the project
  - Why do we iterate?
  - In which phases do we spend the most time
  - Stakeholders in validation
- What activities can we use for validation?: Check that the specification matches the users requirements
- Review and Inspection
  - Informal Review
  - Review Meeting
  - Formal Inspections

- (informal (non-strict)) reviews
  - Can be used i many places and not only requirements validiation
  - 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

  - To help detect and validate requirements, we can model with problem frames
    - 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
    - A well framed problem, might be easier to solve

  - UML
    - Class diagrams
    - Message sequence diagrams
    - State chart diagrams
    - use cases

TODO:

Find noget andet her.

- When the system is being worked on, or is in its end phase, we look at validation and verification (enme 6)

4. Software Quality

Good

Agenda

- 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.
        - **Draw martin fowler**

- Throughout a project, quality management must occur continually

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

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.
- **Software standards**

- Whitebox
    - Unit test
    - integration test
        - Bottom up, bottom down
    - AAA
    - Auto tests
    - Error testing
    - coverage
- black box
    - System testing
    - Boundary analysis
        - Equivalence classes

- ISO 9001 certificate, quality standard
    - Guidelines which companies can choose to follow for quality assurance
    - an audit standard.
    - 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

Quality control

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

- Approaches to quality control
    - Reviews and inspections
    - Automated software assessment
    - Formal inspection
    - Formal verification

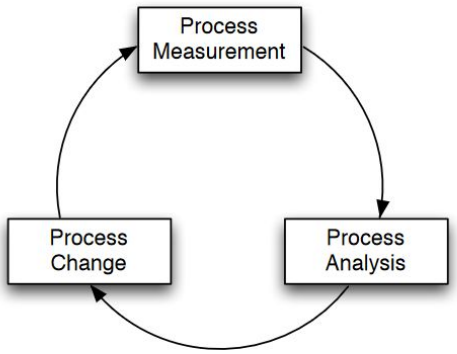
---------- New section ----------

- 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?

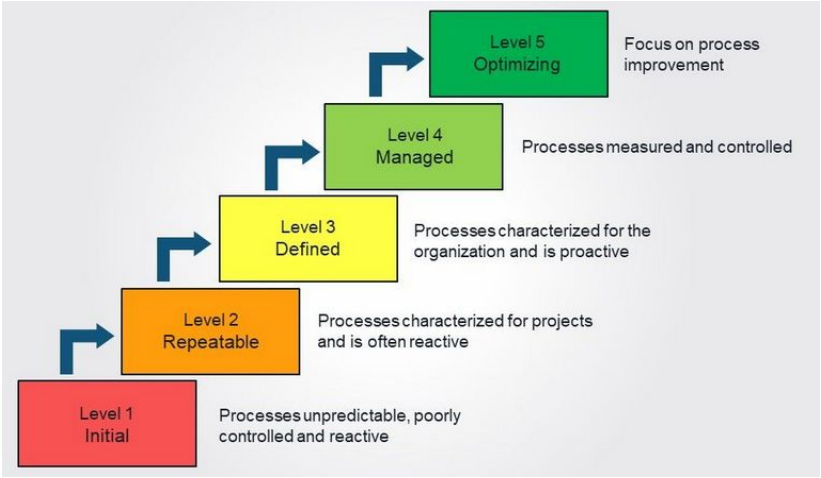
- Process improvement as cyclic activity
    - #Retrospective



- 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

- 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

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



5. Software Architecture

Good

Agenda

- Software architecture is an umbrella term, meaning for a given project, there are alot of software design aspects

- Looking at the first, Architectural design (what is it)?
    - Understanding how system should be organised
    - Designing
        - Overall structure
        - Principle components
        - Relationships
        - How distributed
    - Components relate to functional reqs
    - Design relate to none functional reqs

- 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
        - **Proces 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
        - ***Scenarios***
            - Audience is managers or other stakeholders

- The description of an architecture is illustrated using a small set of use cases

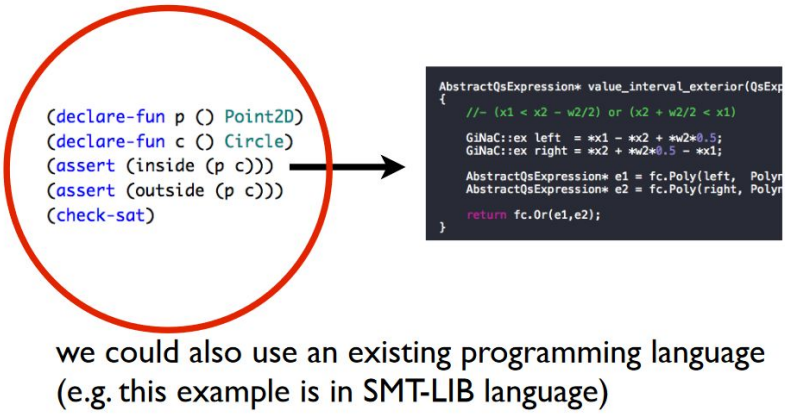
- What is an Architectural Pattern?
  - If we want to solve a non-functional requirement
    - Extensibility, maintainability
    - reusability, availability, security, performance
  - Smart ways to structure components to solve an overall problem.
    - Examples:
      - Microservice - scalability
      - 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

- On a component level, how can we improve productivity?
  - Instead of re-inventing, we use what is already available and made

- Reuse
  - Application framework
    - ASP net (Service hosting, multicore, routing, http/https)
  - Product line → domain specific language

- Reuse-based software engineering- reuse existing software as much as possible
- Different scales
  - System reuse
    - number of applications incorporated into larger "system of systems"
      - Google maps
  - application reuse
    - existing application incorporated into other system (e.g. software product lines)
      - SQL database
  - component reuse
    - entire subsystems or just small modules
      - Nuget packages
  - 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...)

- 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
            - HTML



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

6. Validation and Verification

Good

Agenda

When talking about validation and verification, we need to understand what the two mean

- Are we building the right product? (What the user really wants)
    - validation
    - Does the product satisfy the customer?
- Are we building the product right?
    - Verification
    - Does the product satisfy the specification?

- An overview of testing verification and validation

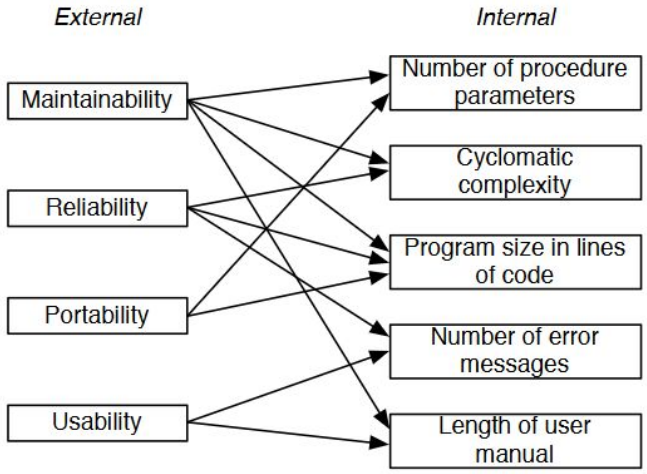
Activity	Verification	Validation
Inspection or review	✓	✓
Unit testing	✓	
Integration testing	✓	
System testing		✓
Acceptance testing		✓
Regression testing	✓	✓

What are the objectives of testing?

- Find errors
- Check for compliance with requirement
- Check performance
- Show absence of defects? (you cant)
    - "Program testing can be used to show the presence of defects, but never their absence"

External and internal software attributes

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



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
        - Un-initialized vars
        - Lines per function
        - args per function etc.
        - Cyclomatic complexity

Planning software tests

- Device a test plan
    - **Testing process:** describe major phases
        - We want at minimum 90% coverage
        - All tests are be done with AAA pattern
    - **Requirements traceability:** ensure test of requirements
        - All requirements, implemented as features, have been tested and can be proven and traced
    - **Test items:** specify artefacts to be tested
        - Which artifacts should be tested?
            - Fx. we use a library, but it has been documented and tested
    - **Testing schedule:** integrate into development schedule
        - When do we test and how?
            - Dedicated tester on team
            - Build server
            - nightly builds
    - **Test recording procedures:** how test results are achieved
        - logging results

What should a test measure

- Achieve an acceptable **level of confidence**
    - Number of defects found in a given series of tests
    - "less than 10 defects discovered in the last 7 days"
- Confident in that the system **behaves correctly**
    - Derived from requirements
    - Compare test results with expected results
- And under all **circumstances of interest**
    - Test student registration system with
        - 10.000 students
        - 12.000 students
        - 15.000 students

Testing methods

- 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

<u>We need to be systematic about testing!</u>

- 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
    - Fx. Arrange, Act, Assert (AAA. Pattern)
    - 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

    - System testing
        - Done by the team
        - Does it work as we intended
    - Acceptance testing
        - Done by the customer
        - Does it work as they wanted
    - Regression testing
        - Unittest, integration test, other test - should be runnable on test server

7. Reuse

Good

Agenda

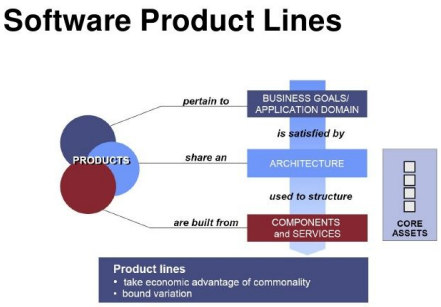
- In the context of software engineering, we want to reuse artifacts which are already made
    - " Reinvent the wheel

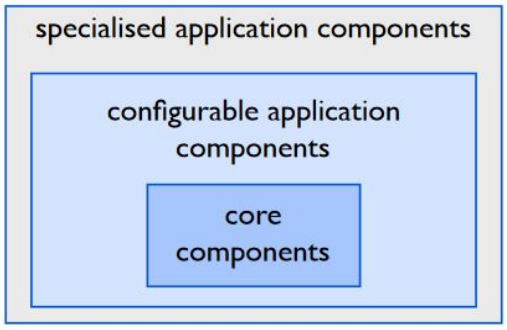
- Forms of reuse:
    - Reuse-base software engineering- reuse existing software as much as possible

- Different scales
  - **System reuse**
    - number of applications incorporated into larger “system of systems”
      - An Example could be the google maps system.
        - Small pay for time saved.
  - **application reuse**
    - existing application incorporated into other system (e.g. software product lines)
      - SQL, mongo
  - **component reuse**
    - entire subsystems or just small modules
      - Packed into nuget packages and pushed to local artifactory, where other devlopoers can fetch
      - Bad to keep of different version
      - If reused, better tests can be assured
  - **object function reuse**
    - single functions, standard libraries offer this
    - in practice re-using specific components tricky
  - **concept reuse**
    - Architectural patteren
      - Microservices example
    - .design patterns
      - Strategy pattern example

- **Application Frameworks** -
  - Examples
    - .NET core web API → crossplatform, serves http/https hosting, handles multiples clients concurrently
    - WPF → User Events, I/O, UI rendering
  - idea: reuse software at higher level of abstraction.
  - application framework: provides a generic structure of an application

- **Product Line** - Multiple similar products. Printers are similar. Share codebase. Adapt for each model in the product line
  - When company needs to support many similar (but not identical) systems
  - Every printer has its own control software
  - Control software very similar
  - So make core product - adapt for each printer
  - Very important that the products have similar constraints. They have common architecture + shared components + some specialisation
  - *Tegn denne*





- 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

- Domain specific language (DSL) - Example Gherkin
    - Restricted programming language to make “saying things” in your domain easier
    - in contrast with general purpose programming language like C++

8. Safety Critical Software

Good

Agenda

- When talking about critical software as a whole, we talk about 3 main categories

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

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

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

Safety in system development - example Pacemaker, Fly, Tesla

- 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**
    - Auto test
    - Code coverage of 100%
    - Promote **uniformity of approach** between different teams
    - Provide **guidance** on design and development techniques
        - Requiriments for programming language

Programming language for critical systems

Common software faults

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

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

    - Virtuel handling .net and java
- Java is therefore a no-go
- There for use ADA!

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

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

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

- **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

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

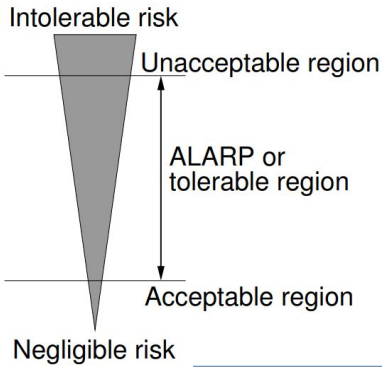
RISK CLASSIFICATION (IEC 1508)

	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



Ethical considerations

- When is a risk tolorable

9. Change Management / Configuration Management Good

Agenda:

As a start, what is management as a whole?

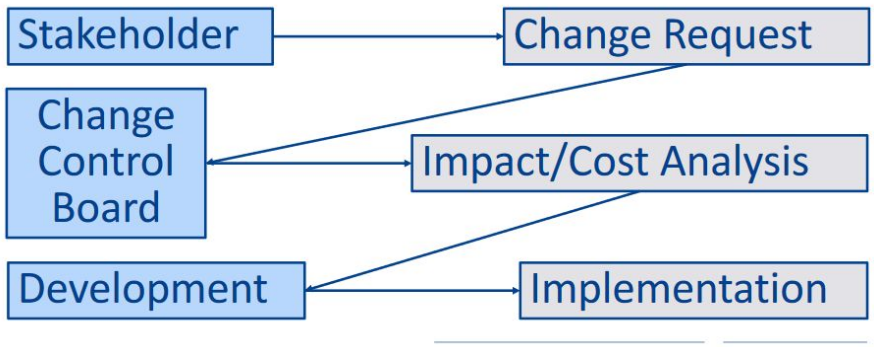
- Managing the project
    - Keep track of change request and the impact of these
    - Keeping track of software components and their version
    - Keeping track on what customer have in production
        - In case of upgrades what is the risk?
    - Keeping track of what were are actually releasing

Configuration management planning

1. Define **What Is to be managed**, the configuration items, and a scheme to identify them(e.g. by their path in the file system)
2. Determine **Who is responsible** for the configuration management procedures
3. Define the configuration management **policies** to be followed
4. Specify the **tools** to be used for configuration management and the **process** for using these tools
5. Describe the **structure of the configuration database** that is used to record configuration information
    a. Which companies uses which version
    b. Which operating system they use
    c. Platform
    d. Known issues

6. **Change management**
    a. Keeping track of requests for changes
    b. working out the costs and impact of making these changes

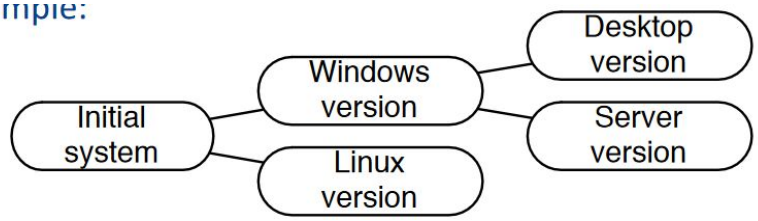
- Typical change management flow:
    - Stakeholder → change request → change control board → Impact/cost analysis → Development → implementation



7. **Version management**
    a. Keeping track of the multiple versions of system components
    b. Ensuring changes made to components by different developers do not interfere with each other

- Software may be produces
    - For different computers
    - For different operating systems
    - For different customers with client-specific functions

xample:



How to versionize:

- Semantic versionizing 1.0.0
- Eks- Git

8. **System building**
    a. The process of assembling program components, data and libraries and then compiling and linking these to create an executable system
        i. Building services packages
            1. Storage
            2. Reusage
            3. Semantic versioning
            4. Private artifactory
            5. Git
                a. How do multiple developers work on same feature
            6. Build pipelines
9. **Release management**
    a. Preparing software for external release
    b. Keeping track of the system versions
    c. A release may contain, for instance:
        - The executable software
        - Configuration files (defining how the release should be configured for particular installations)
        - Datafiles (that are needed for successful system operation)
        - An installation program
        - Electronic and paper documentation
        - Packaging and associated publicity

Important factors influencing the release strategy

- Technical quality of the system
- Platform changes
- Competition
- Customer change proposals