TDDC88 – Sammanfattning

Didrik Grip – i12didgr

Requirements

"Software requirements express the needs and constraints placed on a software product that contribute to the solution of some real-world problems." - Kotonya & Sommerville

Requirement engineering is about clarifying the requirements to avoid misunderstandings and specify what is to be delivered in the end of the project. To avoid misunderstandings, it's important to use complete sentences and use modal verbs such as *shall*, *must* and *will*.

Elicitation – Understanding the *true* needs of the customer

Can be obtained either internally from the projects goals and strategies or externally from environments, stakeholders or a customer. Some of the techniques are interviews, scenarios, prototyping or observations of current use. The trick is to understand what the customer *really* needs and not only what they explicitly say they need – *probe thinking, understanding why the customer answers as they do*.

"If I'd asked my customers what they wanted, they'd have said a faster horse." – Henry Ford

Analysis – Classifying and resolve conflicts between requirements

We classify the requirements in different classes with respect to e.g.

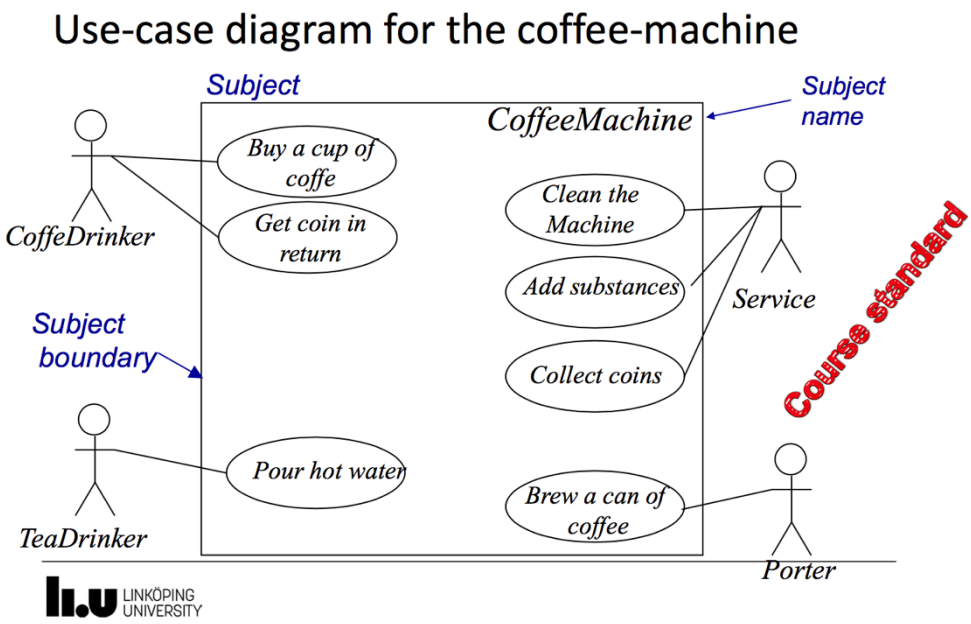
- Functional vs. Non-functional
- Source
- Product or process requirements
- Priority
- Scope – Affected components
- Stability

This is often accomplished with ***conceptual modeling,*** like use-cases, class-models or ER-modeling***:***

Use-case modeling

"a particular form or pattern or exemplar of usage, a scenario that begins with some user of the system initiating some transaction of sequence of interrelated events." – Jacobson et al.

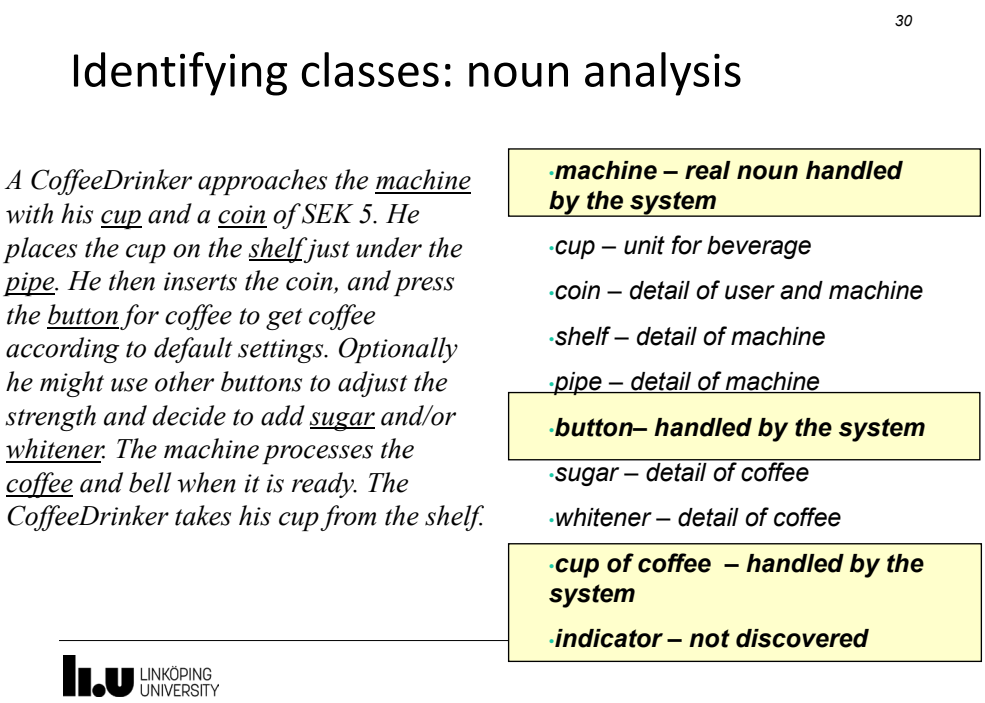
28



A use-case diagram

A use-case is a specific activity, the circles above, and is often accompanied by a description. Example of a use-case where the nouns are being analyzed and some are set as required classes:

30

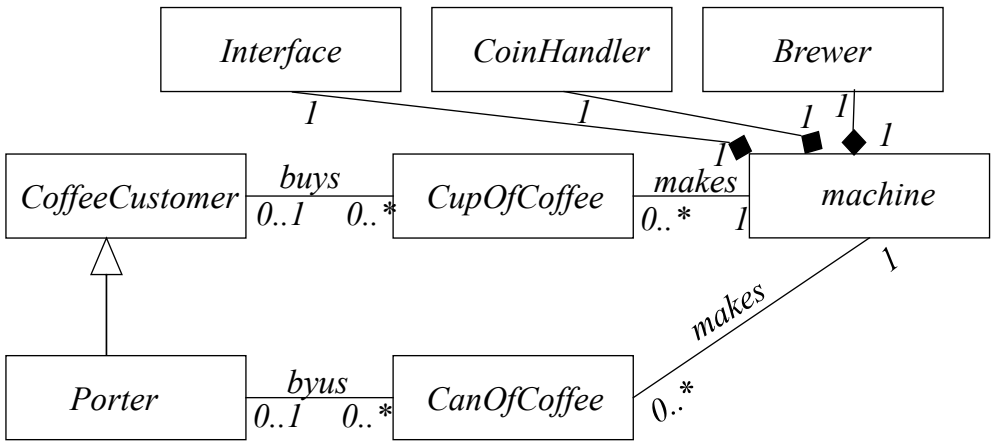


A use-case with classes analysed

Class model

This leads us to the class model, being drawn from the earlier use-case:

The coffee machine class model

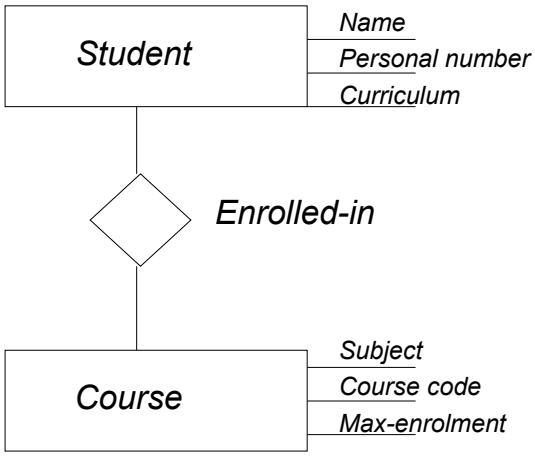


Class model of the earlier use-case

ER-diagram

When it comes to required data, a good model is a classic ER-diagram as is often used in database modeling:

Data model: ER-diagram



Example of an ER-diagram

Specification

SRS - IEEE Std 830-1998

Specifies a good ***SRS – Software Requirements Specification***, with the basic issues:

- *Functionality,* what the software should do
- *External interfaces,* how it interacts with e.g. people and other soft- or hardware
- *Performance,* speed of functions etc.
- *Attributes,* portability, maintainability, security etc.
- *Design constraints,* language, policies, required standards or other limits

An SRS should, according to IEEE 830, be correct, unambiguous, rankable, verifiable modifiable etc. It should only concern the requirements that the software shall meet and every requirement should be open for only one interpretation. It's also important that the SRS ranks the requirements by stability and necessity.

The SRS should NOT however go into design details such as partitioning into modules or describe flows of information.

Pros:

- Good checklist
- Many ways to adapt organization
- Many ways to detail requirements
- You don't need to have everything

Cons:

- Takes some time to read and understand
- Very general, needs to be tailored
- Is no guarantee for a good SRS

Characteristics of good requirements

- ***Unitary (Cohesive) –*** The requirement addresses one and only one thing.
- ***Complete –*** The requirement is fully stated in one place with no missing information.
- ***Consistent –*** The requirement does not contradict any other requirement and is fully consistent with all authoritative external documentation.
- ***Non-Conjugated (Atomic) –*** The requirement does not contain conjunctions. E.g., "The postal code field must validate American and Canadian postal codes" should be written as two separate requirements: (1) "The postal code field must validate American postal codes" and (2) "The postal code field must validate Canadian postal codes".
- ***Traceable –*** The requirement meets all or part of a business need as stated by stakeholders and authoritatively documented. It specifies no more and no less than what is required.
- ***Current –*** The requirement has not been made obsolete by the passage of time.

- ***Unambiguous –*** The requirement is concisely stated without recourse to technical jargon, acronyms (unless earlier defined) etc. It expresses only facts and not opinion. It is subject to only one interpretation. Negative statements are avoided.
- ***Specify importance –*** The requirement must specify a level of importance.
- ***Verifiable –*** The implementation of the requirement can be determined through basic possible methods such as inspection, demonstration, testing or analysis.

User stories

As a (actor) I want (something) so that (benefit), used a lot in the Scrum methodology.
Example: *"As a student I want to buy a parking card so that I can drive the car to school."*
Priority: 3 Estimate: 4

Validation (Formalization)

Some means of validating requirements:

- Prototyping
- Simulation
- Software Reviews
- Model checking
- Formal proofs
- Acceptance testing

Word list:

- ***Functional requirements –*** Describes the functions that the software is to execute, can be easily tested by giving input and controlling the output. Think f(x) = y.
  Example: "The user shall be able to add an item to the shopping basket."
- ***Non-functional requirements*** can be *design constraints* like languages or *quality measurements* such as response times. The latter is also called ***performance requirements.***
  Example: "The minimum response time is 2.0 seconds" or "The system shall be written in Java"
- ***Feature –*** An USP, a distinguishing characteristic of a system item. Like a SMS delivery notification service.
- ***RAM –*** Requirements Abstraction Model, utilizing levels of abstraction. Up to product level and down to function level.
- ***Actor –*** A user of a system in a ***use-case***, can be human or a system.
- ***Stakeholder –*** "an individual, group, or organization, who may affect, be affected by, or perceive itself to be affected by a decision, activity, or outcome of a project"
  Example: Project leader, management, customer, user group, sponsors etc.

Project planning and processes

"A project is a temporary endeavour undertaken to create a unique product or service", it consists of a starting point, at least one item happening on the way and a goal. There is always a balance between goal and process, either you have a clear purpose or goal but no predefined process to follow or you have a strict process with an unclear goal.

SMART goals

Specific – Straightforward, **what** will you do and **why** is it important?
Measurable – If you cannot measure it how do you then know if the goal is reached?
Agreed upon – Agreed upon by all stakeholders
Realistic – Possible with current resources, knowledge and time. Willing and able to do it.
Timely – A clear time frame

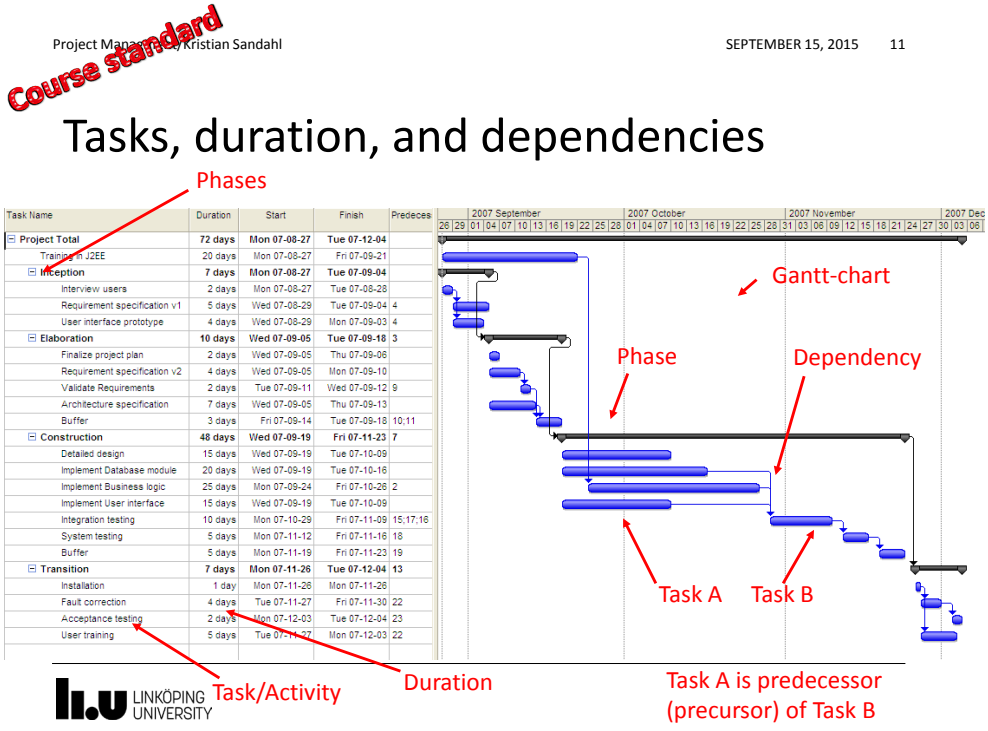
Dependent project parameters

The most important four parameters to consider when setting the goals for a project are:
Calendar time – How much time do we need? When can we finish?
Resources – What resources do we have? Personnel, knowledge, computing power, budget?
Features – What are our features that we want to implement?
Quality – What quality do we require on the final product?

These four are highly dependent and planning a project means to take these into considerations and create a trade-off between them.

GANTT-chart

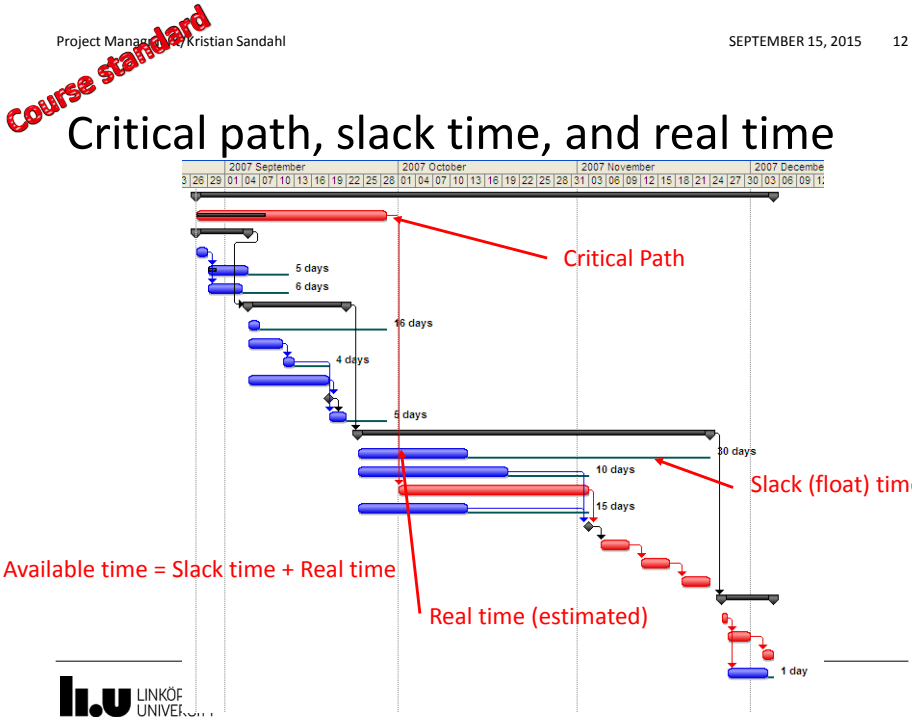
A GANTT-chart is a tool to visualize the process, creating tasks with a duration and dependencies to create a chart where we can prioritize and make sure the project gets finished on time.



Example of a GANTT-chart, making it easier to visualize the process and time approximation.

Critical path

The critical path consists of the activities with the most time-sensitive dependencies. If any of these activities gets postponed, the complete project falls behind in time. It is therefore critical that the activities on the critical path is finished on time.



The critical path can be fetched from the GANTT-chart and displays the activities that are most critical to the project finishing on time.

Effort estimation

Delphi – Experts make individual predictions and show them anonymously for discussion, repeat if not converging.

COCOMO – An algorithmic formula where a numbers of factors are estimated using data from earlier projects. *Example:* Input: Lines of code – Output: Effort (Time)

Planning poker – Agile estimation – Variant of the Delphi method where you have cards marked with units like hours or "effort points" in a Fibonacci-series. You then place cards upside down, flips them and discusses.

- Remember to include **buffer time**!

Risk management

Types of risk:

General risk
Example: "A team member gets sick" "The project gets delayed"

Direct risk
Where the project has great control.
Example: "Our system will not scale"

Project Specific risk
Example: "Anders needs to visit his family since his father is sick" or hardware issues.

Indirect risk
Where the project has little control.
Example: "The servers fail due to an earthquake."

General steps for risk management:

1. **Risk identification –** *"What can go wrong?"* Brainstorming to find possible risks.
2. **Risk analysis –** *"How bad is it?"* Prioritize risks after magnitude
    a. **Probability** – Low, moderate, high, very high (1-4)
    b. **Impact** – Insignificant, tolerable, serious, catastrophic (1-4)
    c. **Risk Magnitude Indicator** = Probability x Impact
3. **Risk planning – "***What do we do if it happens?"*
    a. **Avoidance –** Reorganize so the risk disappears.
    b. **Transfer –** Reorganize so someone else takes the risk.
    c. **Acceptance**
        i. **Mitigate –** Lower the probability.
        ii. **Contingency plan –** Lower the impact.

The project plan

Is a tool for the project manager to get a clear overview of the process and goals. It's also a communication medium between stakeholders and specifies *what* should be done, *when* and by *who*.

Contents of the project plan:

- ***Description***
    - Background, constrains, goals etc.
- ***Organization***
    - Roles, available knowledge or skills, training plans, communication.
- ***Time and Resource Plan***
    - Milestones, tollgates, deliverables, activities, resources.
- ***Risk Management***
    - Risks – probability and impact.
    - Mitigation and contingency plans.

Common roles in a software project

The ones applied to our current project is marked in ***italics***

- ***Project manager –*** Ensures that the plan is being followed and goals reached.
- ***Product manager***
    - **Strategic (Product owner or sponsor) –** Market communication and analysis, budget responsibility, decides features
    - **Operational –** Technical management and expert, effort estimation
- ***Configuration manager –*** Selects and maintains tools, decides on what's in a release
- ***Line manager –*** The legal employer, ensures competence development and a good working environment.
- ***Process manager –*** Decides on processes and adherence to these
- ***Analysts –*** Handles everything that has to do with requirements, writes SRS
- ***Architect –*** Ensures that requirements are met and specifies a high-level architecture
- ***Lead designer –*** Handles prototyping and design issues not covered in architecture, also designs the UX and dialogue etc.
- ***Environment manager –*** Creates and maintains the environments for development and testing
- ***Developer –*** Develops the system

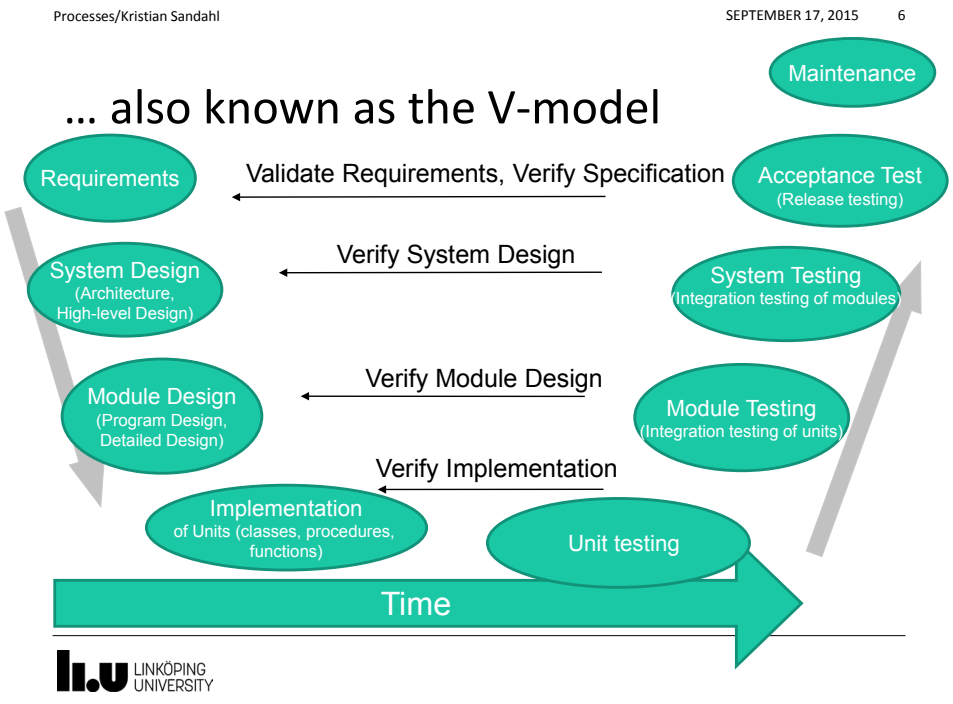
- **Procurement responsible –** Buys components and licenses
- **Component adaptor –** Adapts external or reused components to the system
- **Integrator –** Puts the pieces of the software together to a system
- ***Testers* –** Tests and evaluates requirements
- ***Quality coordinator* –** Measures quality and organizes reviews
- ***Deployment manager* –** Ensures that the product is installed and made available
- ***Technical writer* –** Responsible for documenting
- **Course developer and leader –** Creates and preforms training material and courses
- **Helpdesk –** Helps with and documents customer issues
- **Operations manager –** Ensures that services are provided to the customer
- **Systems engineer –** Performs maintenance and monitoring on active systems
- **Librarian –** Manages component library and identifies reusable components
- **Document responsible –** Decides on standards and database modeling

Processes

Processes are a way to reach the goal, they are an ordered set of activities where each activity has entry and exit criteria and some constraints.

V-model

The V-model is ordered by abstraction and time, where we often start and end in a high level of abstraction and implement in the middle.



An example of the V-model where we have a level of abstraction on the Y-axis and time on the X-axis

Waterfall

If we remove the Y-axis, that is the abstraction level, from the V-model we get the classical waterfall model. This is one of the oldest models for development and includes a strict constraint that every phase is to be completely finished before moving on to the next.

Problems with the waterfall model includes changes during the process being hard to implement due to the strict order of phases. Also feedback is hard to implement and we need to have a really good time estimate for the phases to finish on time.

Advantages with the waterfall model include its simplicity and easiness to understand. Can be applicable to short projects or stable projects where the requirements are not expected to change e.g. fixed-price contracts.

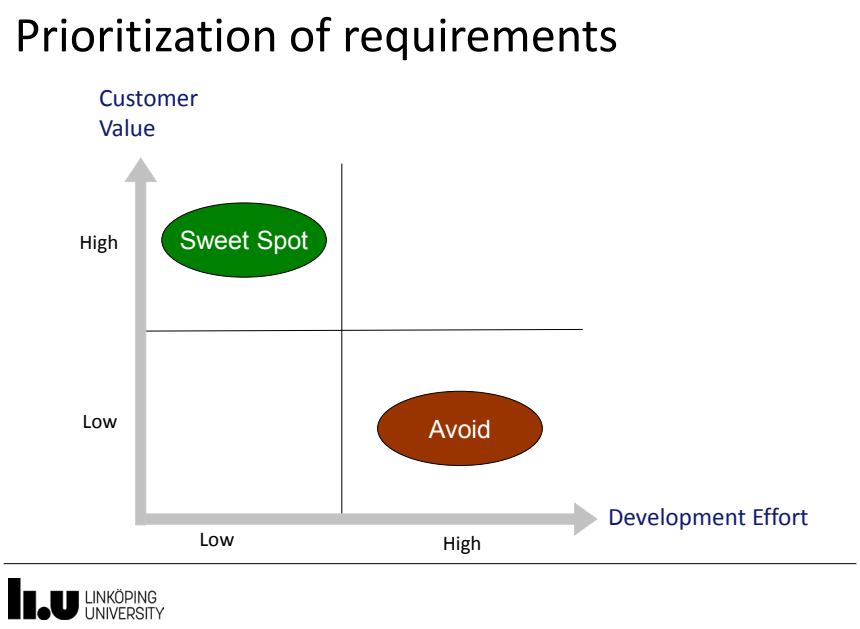
Iterative

"If the requirements change, why don't we do it again?" Royce said in 1970 and marked the birth of the iterative model. In this model we utilize either waterfall or v-model and apply it to a number of iterations where we get feedback in the end of each. Here we fix the two project parameters of *time* and *resources* and revisit *features* and *quality* to update these to current needs. We can also call the model of adding features in different iterations an **incremental model**, where you keep on adding to what you have to get closer to the goal.

Problems with the iterative model include the problems with mapping requirements to different iterations and prioritizing the features, some might be forgotten.

Advantages is mostly about the flexibility and the spread out workload of the members. The team can continuously improve the process and misunderstandings are made clear early.

In an incremental model the prioritization of requirements is important, a comparison between **customer value** and **development effort** is often used as a guideline.

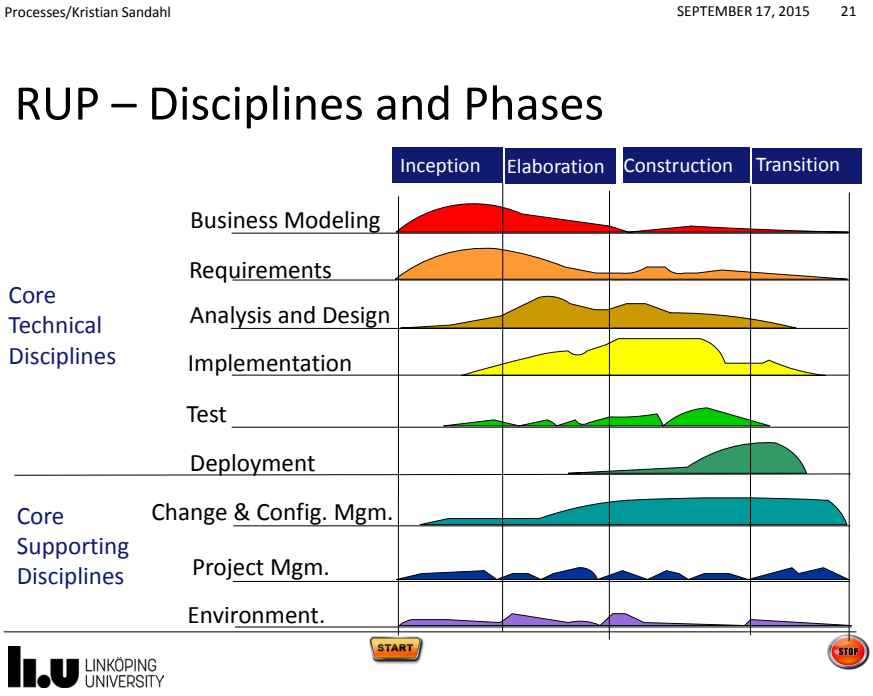


Example of a chart with customer value and development effort.

RUP - Rational Unified Process

RUP is an iterative software development methodology defined in 1997. It has some scaled down versions including Open and UP with agile components. It visually describes the workload of different areas and has four main phases that consists of:

Inception – Requirement specification, prototyping
Elaboration – Architecture, project plan
Construction – Development and testing
Transition – Delivery to customer



An example of a visual RUP-representation

Agile development methodologies

Agile development includes a couple of values that includes agility to changes in the planning and having a working software rather than comprehensive documentation. It also includes a grade of customer participation rather than having fixed contracts. A number of methodologies fall under the agile umbrella, such as:

eXtreme Programming (XP)

XP is a methodology that describes in detail how the software should be implemented, some of the main elements are pair programming, extensive code reviews, test driven development and a flat organization with frequent communication between personnel. Some critics claim that XP is too unstable and lack overall design and documentation.

Scrum

Scrum defines a methodology where cross functional teams develop a product in a couple of iterations, also called **sprints.** It includes a couple of roles such as a **product owner** and a **scrum master**. The typical usage of scrum is to first use a **product backlog** to write **user stories** and from these define a number of tasks to spread out over the sprints in a **sprint**

backlog. Also frequent **scrum meetings** and an extensive **review** and **retrospective** at the end of all sprints to define lessons learned.

Kanban
Kanban is a visual system deriving from the Japanese word for billboard (看板). At the centre of the methodology is a billboard where feedback is posted for the management and taken into consideration when implementing changes throughout the process. One of the main principles are *leadership at all levels*, making everyone responsible for changes in the process.

Lean Software Development
An adaption of the lean principles for software development, often favoured by start-up companies trying to penetrate the market. Some principles are to *eliminate waste, decide as late as possible* and to *empower the team*.

Word list:

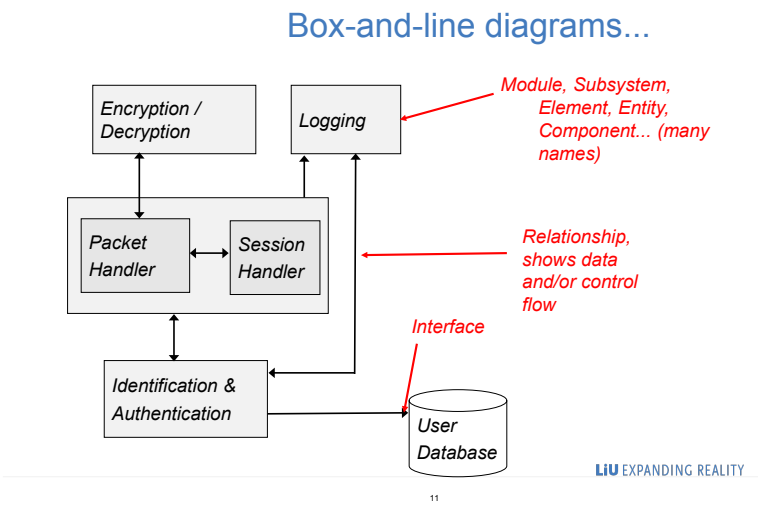
- ***Task/Activity –*** A task that is to be finished in the progress for the goal, is represented in the GANTT-chart.
- ***Phases –*** A gathering of tasks into a phase makes overall views of the project easier. Often ends with a milestone or a tollgate.
- ***Milestone –*** A predefined sub-goal during the project that leads one step closer to the final goal. E.g. an alpha version or tests finished. Should be defined in SMART.
- ***Tollgate –*** An external decision point where a stakeholder or other can abort the project or choose to continue. E.g. a first prototype or inspection.
- ***Slack time –*** The amount of buffer time before the task affects other tasks and the overall time for the project.
- ***Real time –*** Estimated time for the activity / project.
- ***Available time –*** Slack time + Real time.
- ***Status reports –*** A summary of the current status, what has has happened since the last report and what should happen next. It also details current problems and risks and should be compiled regularly throughout the project.
- ***Brooks' law -*** "Adding manpower to a late software project makes it later" because of e.g. training
- ***Time box –*** A fixed time period to a planned activity, deliverables at a deadline
- ***Software life cycle –*** The life cycle of the software, from specification to implementation to testing to maintenance.
- ***Product backlog –*** A requirement list for a scrum team.
- ***Product owner –*** The person responsible for the product backlog in the scrum team.
- ***Scrum master –*** Chairman of the scrum meetings and responsible for planning sprints.

Design and Architecture

The design and architecture of a system is a description of how the system is / will be implemented. A number of ways to do this is discussed below. The design process starts with decomposing the system into modules through communication between stakeholders. An important concept in system design is reusability, to be able to reuse components in multiple modules.

Box-and-line diagram

Prototyping of the system often starts with drawing modules in a box-and-line diagram, shortly describing the modules and the relations between these.



A simple box-and-line diagram of a system.

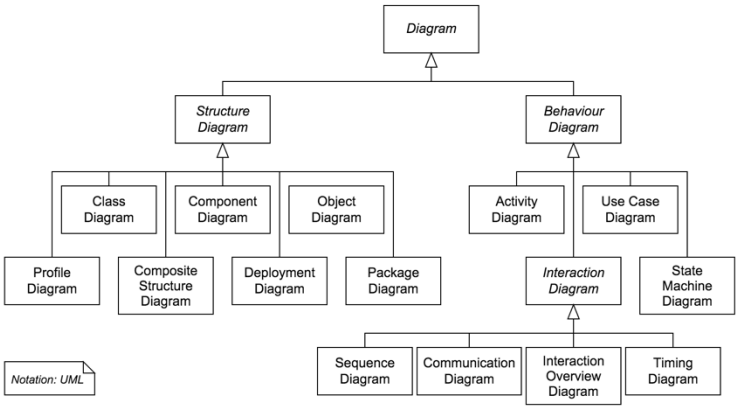
Views

After the box-and-line diagram a number of architectural views are often composed, these describe the system in more detail and is often drawn up using UML. The three views described in this course are:

- **Implementation view**, shows packages, components and artifacts.
- **Execution view**, shows components, connectors and sub-systems.
- **Deployment view**, shows the physical machines.

UML

Unified Modeling Language is a modeling-language in software engineering that visualizes the design of a system. It contains of a number of specified diagrams, shown below, and in this course we specify a number of them.

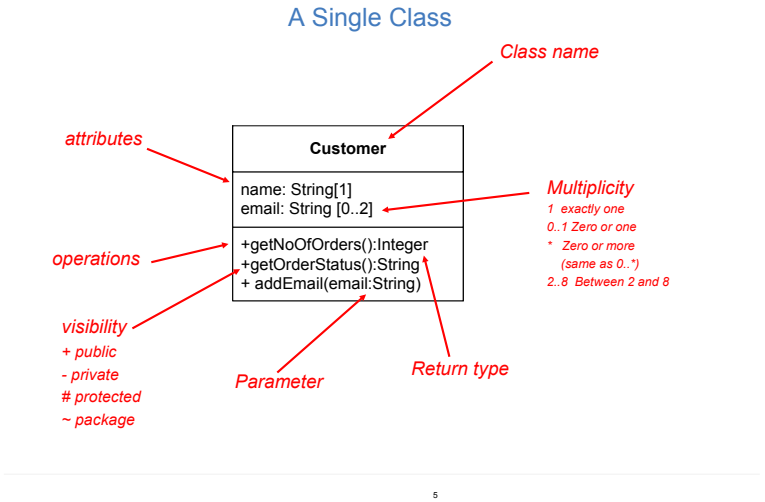


The hierarchy of diagrams in UML 2.0.

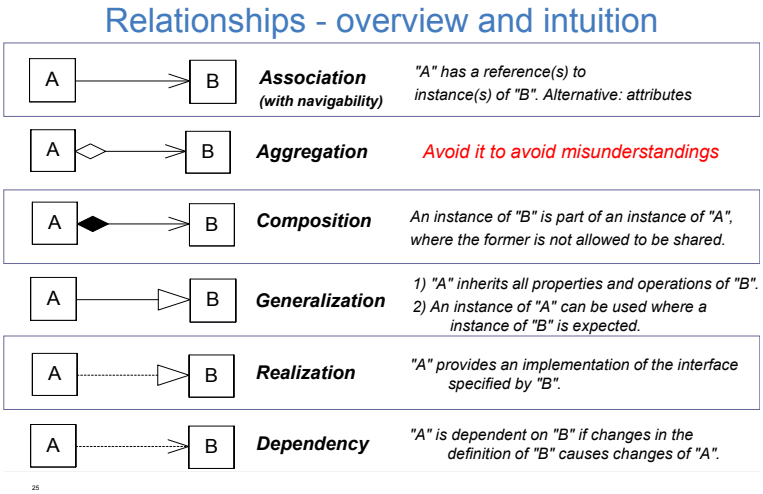
Structure diagrams

Class diagram

A class diagram is one of the most common UML diagrams, it shows classes of a system with it's internal attributes and operations. Classes are also tied together with associations.

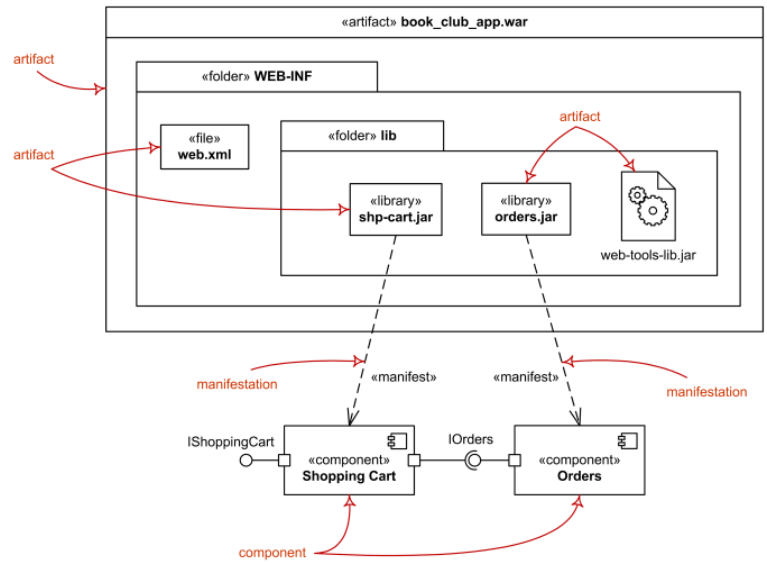


A class with attributes and operations (functions).

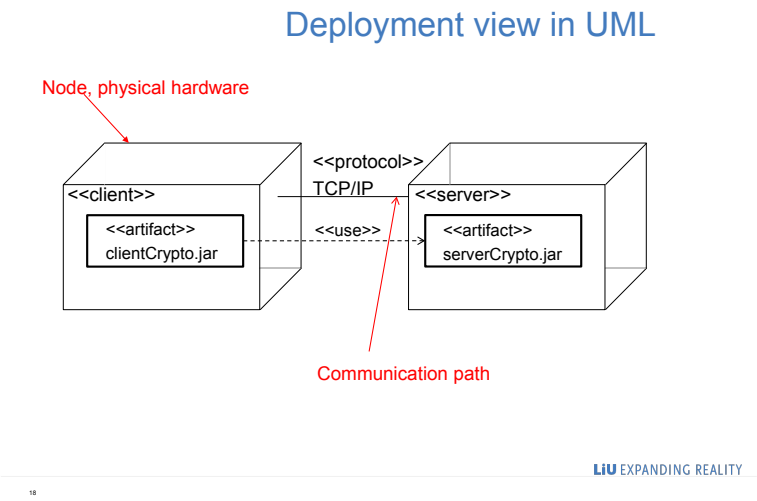


The relationships between classes. For a more detailed description see lecture 6.

Deployment diagram
Consists of artifacts that utilizes components and relationships between these through communication paths to create a overview of the system.



A component diagram consisting of artifacts that utilize the components Shopping Cart and Orders.



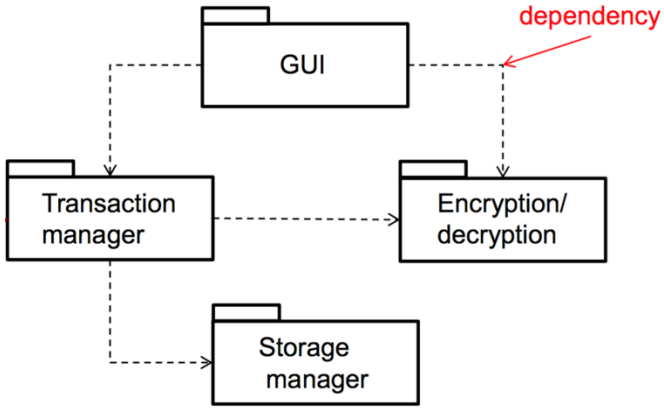
A system with server and a client with cryptography artifacts.

A couple of central concepts of designing the deployment view is:

- ***Coupling,*** *dependency between modules* – we want low coupling due to:
  - *Replaceability*
  - *Enable changes to single modules without affecting the system*
  - *Testability and isolating errors*
  - *Understandability*
- ***Cohesion,*** *relations between internal parts of the module* – we want high cohesion
  - *Easier to understand*
  - *Easier to maintain*
  - *Every part does what it is supposed to, with low cohesion they have nothing in common and are supposed to be in separate modules*

Package diagram

Shows the packages to support the system. To get a view of *what do we need to design and what already exists* and also to know *where things are* for future usage.



Behaviour diagrams

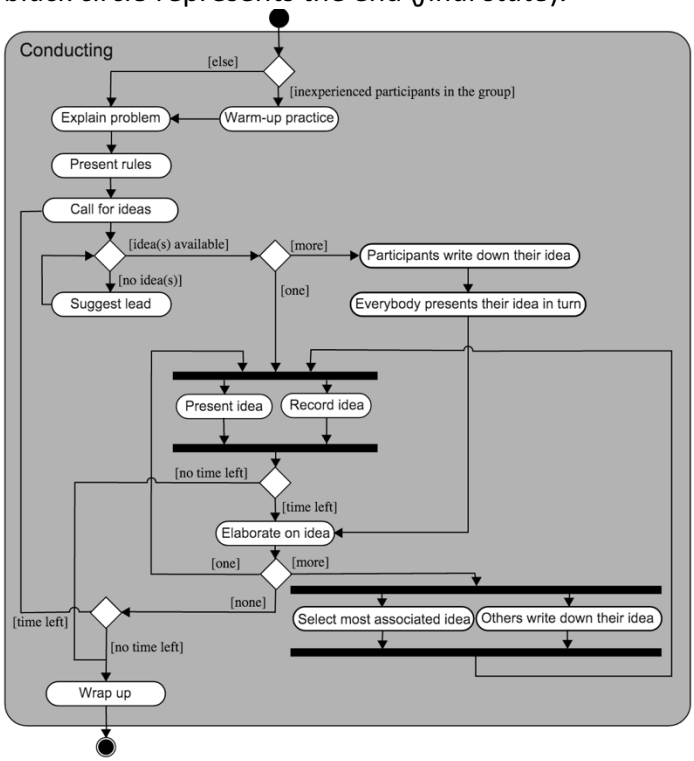
Use case diagram

These have been covered in the previous chapter about requirements.

Activity diagrams

Represent an activity from start to end, like a brainstorming process shown below. The shapes of an activity diagram are:

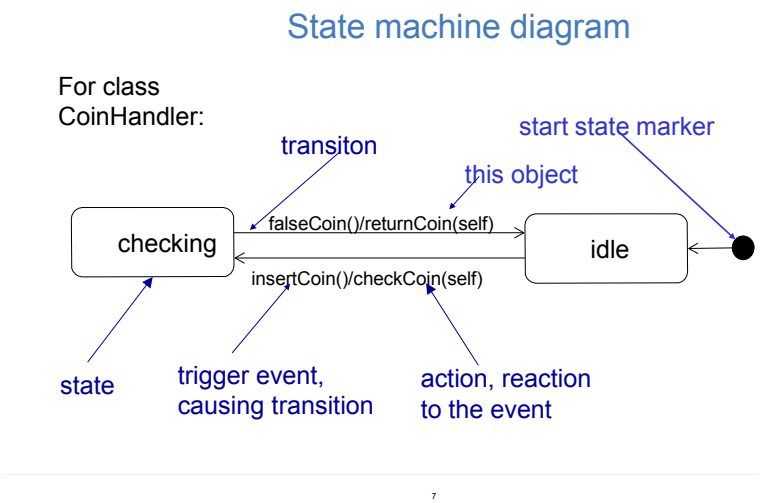
- *rounded rectangles* represent *actions*.
- *diamonds* represent *decisions*.
- *bars* represent the start (*split*) or end (*join*) of concurrent activities.
- a *black circle* represents the start (*initial state*) of the workflow.
- an *encircled black circle* represents the end (*final state*).



An activity diagram of a brainstorming process.

State Machine diagrams

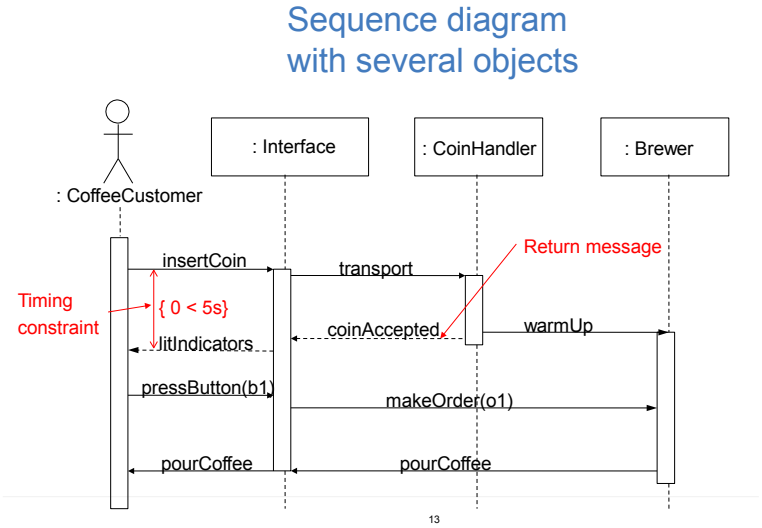
Quite similar to the activity diagram but represents current states and actions between these. A *hollow circle* represents the end, if any.



State machine diagram of a coin handler in a vending machine. Notice no hollow circle, therefore no end.

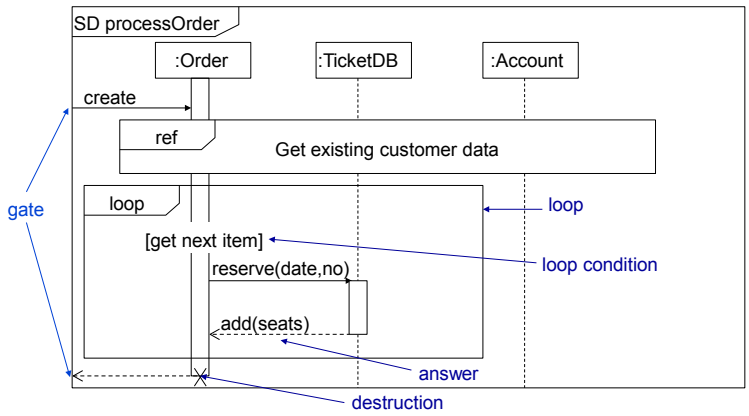
Sequence diagram

A sequence diagram is an **interaction diagram** that shows the operation of processes with one another over time. Blocks represent the timeline of the objects specified at the top.



A sequence diagram for a coffee machine, with the customer as an external actor.

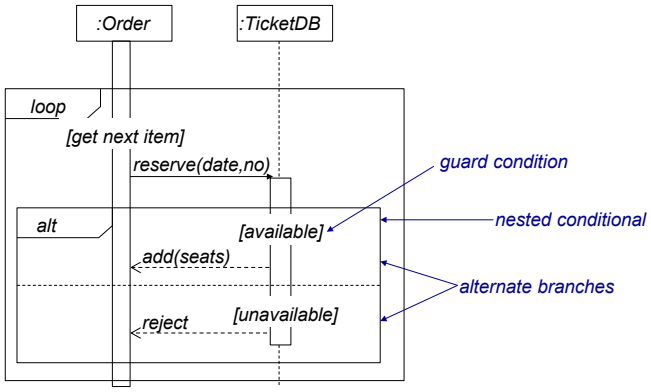
Combining fragments of sequence diagrams



14

Fragments of a sequence diagram can be grouped to enable loops.

More fragments of sequence diagrams



17

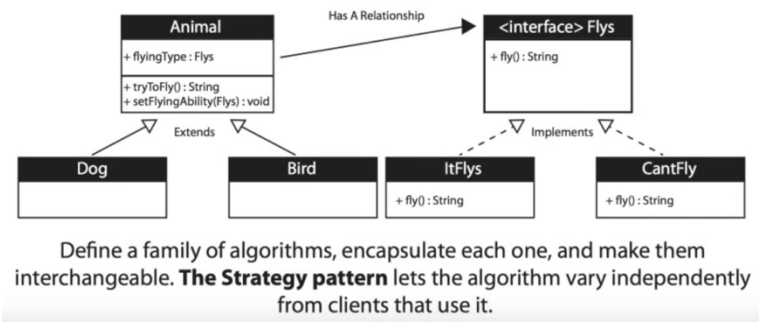
Or conditional branches.

Design patterns

A design pattern is a standard solution to a system design to enable reuse of information and components. A number of patterns are common use, in e.g. the three below.

Strategy

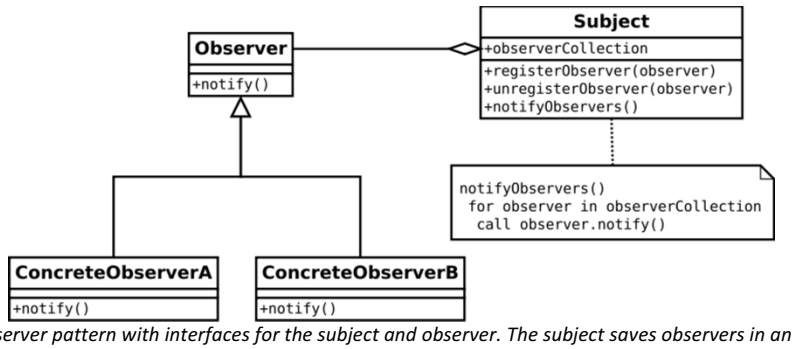
If you're going to implement an object that has a behavior that depends on a number of conditionals. E.g. a superclass of animals and subclasses that can either fly or not, you can make an interface that is Flys that implements IsFlying or CantFly. Then you set the individual animal to either at runtime.



A Strategy pattern with animals that can either fly or not. flyingType is set to either at runtime.

Observer

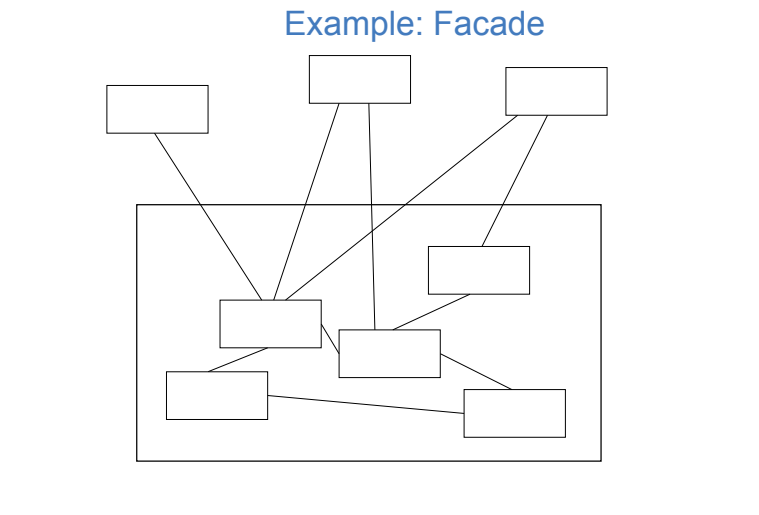
Is a pattern between a subject and one or more observers that need an update every time something changes in the subject. Think of it as a subscription to a stock market (subject) and observers (investors) that gets notified whenever the price changes.



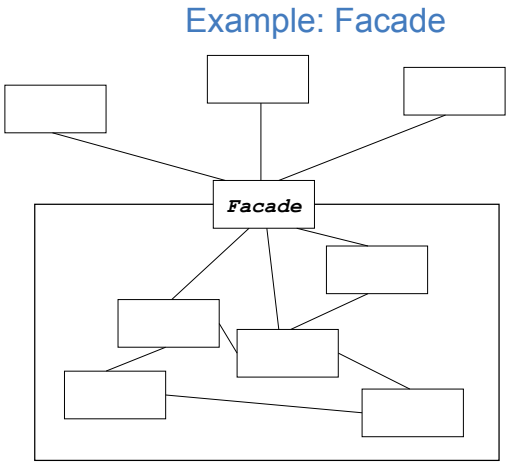
The observer pattern with interfaces for the subject and observer. The subject saves observers in an array.

Façade

A unified interface to a subsystem to make it easier to use, this is to protect the contents of the subsystem and make it able to change parts of it without having to change the entire system.



A system without façade, the subsystem is within the square and external parts are connected where needed.



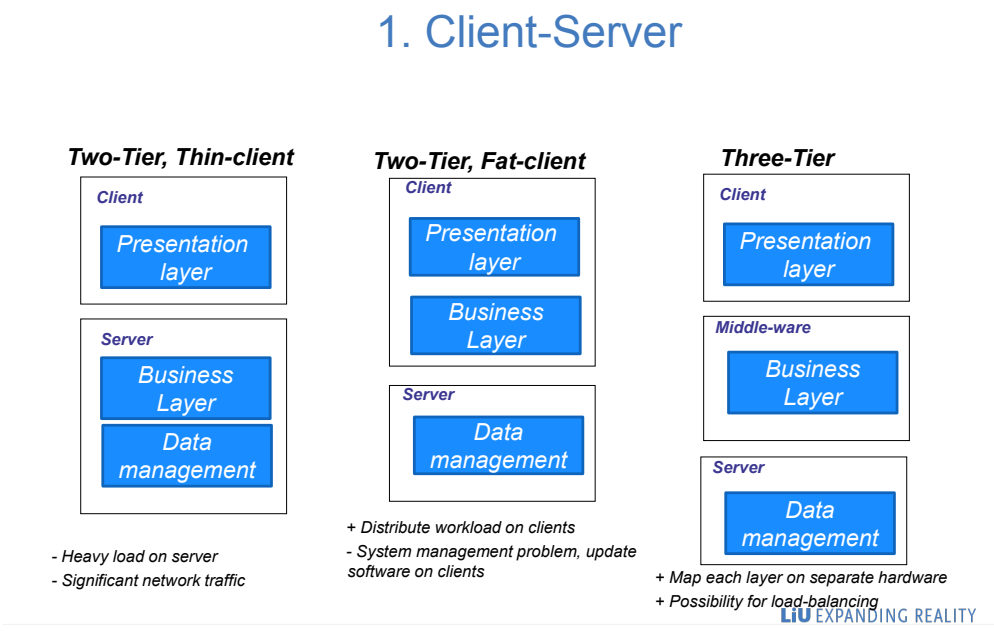
The same system with a façade applied, to enable changes in the subsystem without affecting external parts

Architecture Styles

A number of architecture styles also exists and should be taken into consideration, these are more related to the architecture of a system rather than the design patterns described above.

Client-server

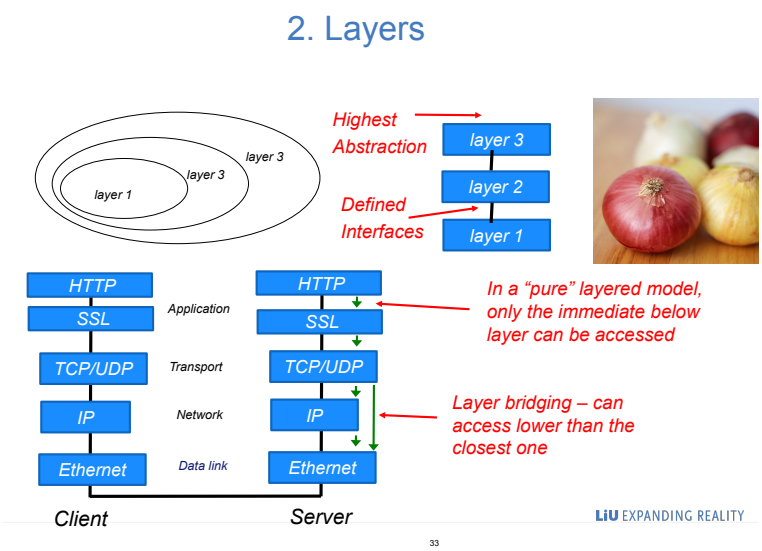
Describes decisions of how much code and workload should be distributed between the client and the server.



Three common structures of client-server architecture, two-tier with fat or thin client and the three-tier structure.

Layered

A layered architecture has a clear structure between high-level to low-level interfaces where communication is only between directly linked layers. Some bridging can be accepted.



A layered architecture.

Pros:

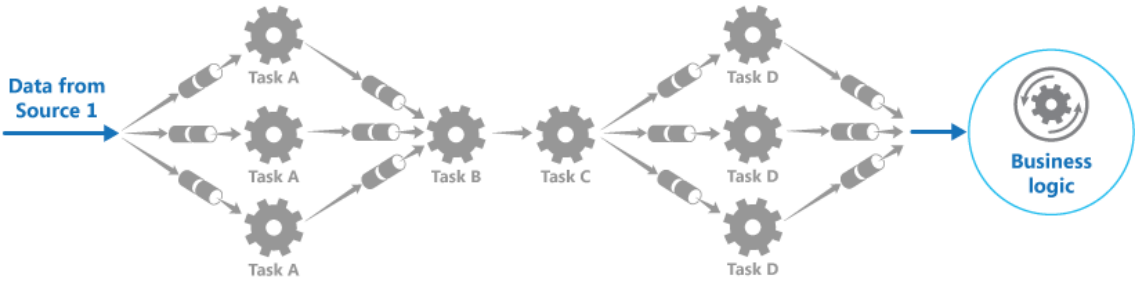
- Easy reuse of layers
- Support standardization
- Dependencies and modifications are kept local
- Supports incremental developing

Cons:

- Could give performance penalties
- Layer bridging overrides modularity

Pipe-and-filter

Consists of decomposing complex processes into filters that perform a process and then sends the data forward though pipes. Think of this as the stages in a pipeline CPU-architecture.



An example of a pipe-and-filter architecture.

SOA - Service-oriented architecture

Consists of decomposing the system into components that provide services for other components via communication protocols. Here we view the system not as a collection of hardware or software but as a collection of services. This is common on most modern web-platforms, a prime example being Amazon that spread out their services on partner companies to enable easy scaling of their system.

Documentation of design

- **System overview**, a brief description stating
    - Who the users are
    - The main requirements and constraints including quality factors
    - Any important background information
    - A “mental” model of the system
- **Structural views,** give an overview and describe each element and all relations.
    - Implementation view
    - Execution view
    - Deployment view
- **Mapping between views,** describe how the views relates to increase understanding.
- **Behaviour views,** to describe behavior for elements by e.g. texts, sequence diagrams or state machines.
- **Rationale,** a motivation to why the design is as it is and what would happen if you change it.

Word list:

- ***Module –*** A decomposed, atomic part of the system.
- ***Prototyping –*** An iterative process of decomposing the system into modules.
- ***Artifact –*** Physical code, a file or a library in a component diagram.
- ***Component –*** The artifact implements a component. Like the artifact “clientcrypto.jar” implements the component Encryption.

Testing and SCM

Testing

After we built the system we need to test it, two central concepts are:
Validation: Are we building the right system?
Verification: Are we building the system right?

Testing is about making sure that we get the correct output from the different modules and that we get the result that we expected from the system. The system is *executed*, the results are *observed* and an *evaluation* is made.

Either the developers do the testing, they have a understanding of the system but the risk is that they would test too gently with fear of not having deliverables at the end of the project. Or, independent testers can learn about the system, they will try hard to break the system and is driven by quality.

Black box testing

Functional testing, given the *input,* the software shall provide the correct *output.* Establish confidence for the complete system, but the system can still contain internal bugs.

Includes three types of testing:

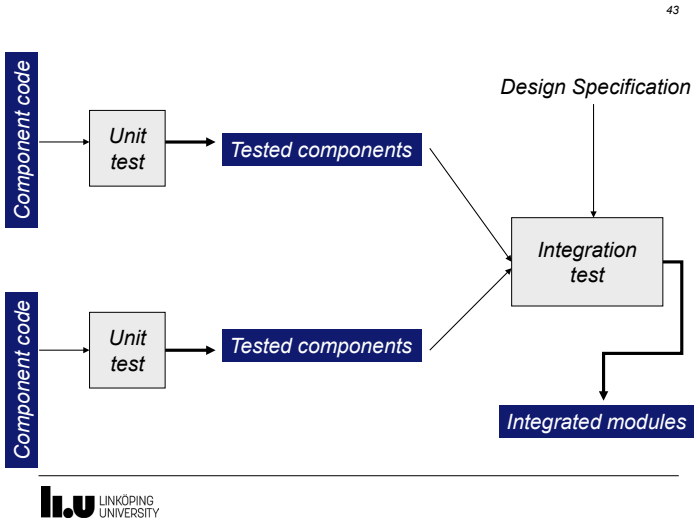
- **Exhaustive testing –** Testing with all possible input variables to the program
- **Equivalence class testing –** Testing all equivalent sets of data, with equivalent meaning that it is treated the same by the system. E.g. if the rule is that if you are 18 years or older you can borrow max 100,000, but no less than 10,000. The equivalent classes are:
    - **EC1:** age < 18
    - **EC2:** age >= 18
    - **EC3:** sum < 10,000
    - **EC4:** 10,000 <= sum <= 100,000
    - **EC5:** sum > 100,000
- **Boundary value testing –** Focuses on the boundaries between equivalent classes. So we test one point on the boundary, one lower and one higher.

White box testing

Seeks fault within the system, trying to find *inputs* and *outputs* so that a certain operation inside the module is executed. Seeks out bugs actively.

Unit testing

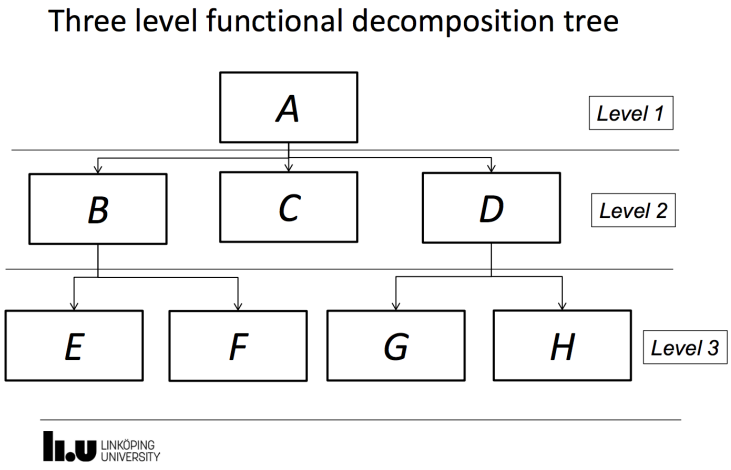
Tests being run when developing single components, before accepting them into the system.



Unit tests are being run on single components, while integration testing takes place before integration into modules..

Integration testing

Are tests run when integrating single components into integrated modules. Have four main strategies:



Each letter represents a component; the complete tree is the integrated module.

Big-bang

Integrates all components in one big-bang, thereafter tests the complete module. Provides difficulty to isolate defects found and had a possibility to miss critical defects, but is quick.

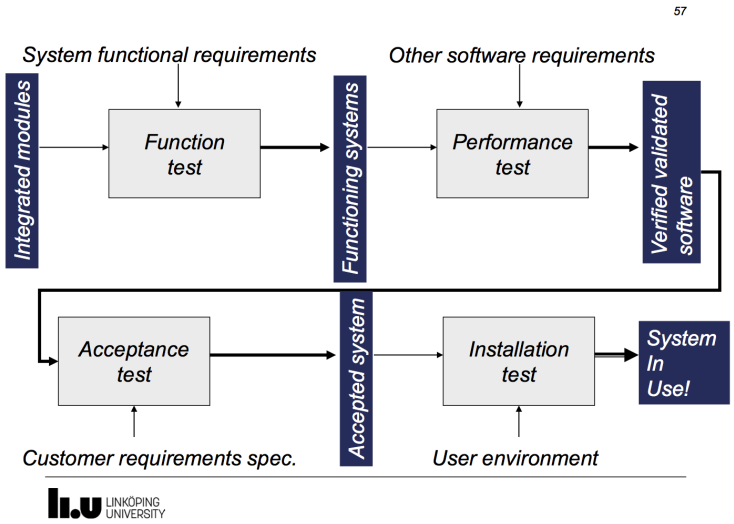
Bottom-up

Tests from the bottom up, i.e. (E, F, B) & (D, G, H) is tested before integrating. Pretend components called **drivers** are used to mimic (A) and test these sub-systems individually. Disadvantages is that we can catch very big interface defects late and we require good drivers for testing. Advantages include easy use of incremental development.

Top-down
Tests from the top down, i.e. (A, B, C, D) are tested before integrating with (E, F) & (G, H). Here we use pretend components called a **stub** to mimic the output of lower level subsystems. Stubs are easier to write than drivers and we can discover general design defects early but we require many stubs and the basic functionality of the module is tested late.

Sandwich
A combination of bottom-up and top-down testing where we test (A, B, C, D), (E, F, B) & (G, H, D) in parallel. Is good if you have a big system with many layers but require more resources and doesn't test the individual subsystems thoroughly.

System testing
Before integrating the modules into the complete system, we need a couple of tests before it's accepted and in use.



The four stages of system testing.

Function test
Tests functional requirements. Testing one function at a time, useful to have an independent test team who knows the expected actions and output.

Performance test
Tests non-functional requirements. Consists of a number of tests based on the requirements, e.g. timing tests, volume tests, security tests, maintenance tests and usability tests. Often run by gathering statistical data.

Acceptance test
Test that the users or customers needs are fulfilled. Consists of:

- **Benchmark tests –** Specially designed test cases
- **Pilot tests –** In everyday work
    - **Alpha test –** At developer's site, in a controlled environment
    - **Beta test –** At one or more customer sites
- **Parallel tests –** Testing the new system in parallel with the previous one.

Installation test
Thorough testing at the customers site, if the beta test has been successfully run at the customers site this test might not be needed.

Typical faults in the system:

- **Algorithmic –** Division by zero
- **Computation & Precision –** Wrong order of operations
- **Documentation –** Documentation doesn't match code
- **Stress/Overload –** Data size (dimensions of tables, size of buffers)
- **Capacity/Boundary –** x devices, y parallel tasks, z interrupts
- **Timing/Coordination –** Real-time systems
- **Throughout/Performance –** Speed doesn't match requirements
- **Recovery –** Power failure
- **Hardware & System Software –** External connections
- **Standards & Procedure –** Doesn't meet organizational standard - difficult for programmers to follow each other.

Termination problem

How do you know when to stop testing? It is influenced by:

- Deadlines
- Test cases with a decided percentage passed
- Test budget depleted
- **Coverage of code**

Full path coverage has been reached when all possible paths though a system has been executed.

Software Configuration Management (SCM)

When developing a large size software, you need to have some sort of system to keep track of changes throughout the process. If one developer makes a change and another developer makes another change we need some sort of system to merge these together. This is where SCM is utilized.

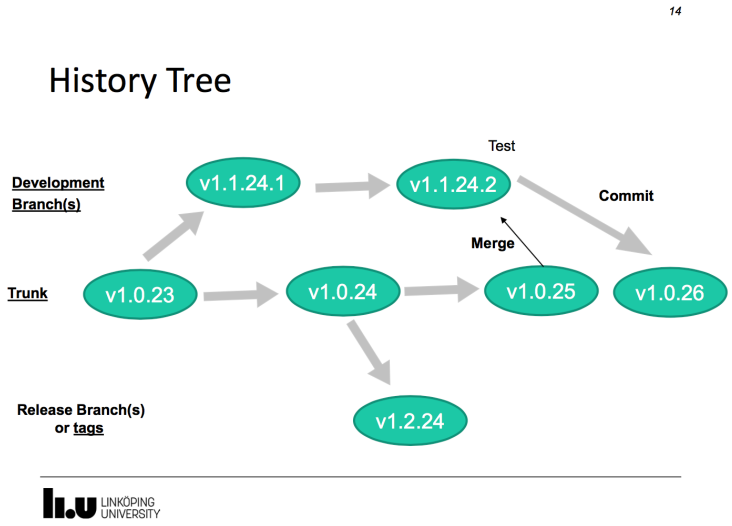
Continuous integration

The practice of continuous integration means that you integrate changes frequently (most often daily), important practices are:

- Automated building of the system
- Automated testing of the system
- The use of SCM systems for integrating code

Revision control

Revision control systems are central in SCM, we have a unified repository with a trunk and a number of independent branches. If you make a local change you merge this with a unified branch and then commits the changes to the trunk.



The versions of a revision control system, with the main trunk and two branches.

Centralized (SVN)

Subversion (SVN) is a common centralized revision control system where the code repository is centralized on a server. You then download a local working copy, make your changes, and merges, solving any conflicts before you commit and upload your changes to the repository. All changes are kept centrally on the server and the local version is just a copy of the current working version.

Distributed (Git)

Git is however a distributed revision control system where the repository is locally on your own computer and the centralized working version is the current one. Here you keep all the versions locally and can commit to it several times before pushing (uploading) to the central server.

Word list:

- ***Error –*** When people make mistakes while coding.
- ***Fault –*** A defect that is the result of an error. **Faults of commission** occurs when we enter something correct into an incorrect representation while **faults of omission** are when we fail to enter correct information.
- ***Failure (anomaly) –*** a failure occurs when a fault executes, i.e. only on faults of commission.
- ***Oracle –*** The one who decides what is a passed test, can either be an expert human or an automated system. E.g. “Values between 0.99 and 1.01 is accepted outputs”.
- ***Regression testing –*** The practice of testing the complete system when a new component has been integrated.
- ***Smoke testing –*** Test only the most critical parts of a system quickly.
- ***Jenkins –*** A tool for automated testing.

- ***Trunk –*** The main working code that is being developed in a revision control system.
- ***Branch –*** A smaller part being branched out from the trunk to implement e.g. a function or a component before integrating into the main trunk.
- ***Merge –*** The process of integrating code into a branch or the main trunk from your local repository.
- ***Commit –*** Uploading local changes to the central revision control system.

Quality

Factors

A number of measurable software **quality factors** exist; these are:

- Correctness
- Reliability
- Efficiency
- Usability
- Integrity
- Maintainability
- Flexibility
- Testability
- Security
- Portability
- Reusability
- Interoperability
- Survivability
- Safety
- Manageability
- Supportability
- Replaceability
- Functionality

Software reviews - Inspection

Software inspection is implemented throughout the process to find defects and improve the process itself. It is a systematic peer examination and *not testing*. Everything from the SRS to design, source code, process descriptions and installation procedures can and should be inspected to ensure quality.

Participants (Roles) of an inspection, should be 2-6 persons:

- **Leader (moderator) -** planning and organizing the process of inspection.
- **Recorder -** documents the inspections defects, results and recommendations. Can be the same person as leader.
- **Reader -** informs the software product and highlights important aspects.
- **Author -** performs rework to meet criteria. Shall not be leader, recorder or reader.
- **Inspector -** identifies and describes defects, can be assigned specific topics. All participants are inspectors.

Process of an inspection:

- **Input –** *responsible: Author*
    - Objective statement
    - Software products or artifacts to be inspected
    - Inspection procedures
    - Reporting forms
    - Known defects
    - Source documents – SRS, descriptions, documentation
- **Plan and overview –** *responsible: Leader*
    - Identifying the team and assigning responsibilities
    - Schedule meetings
    - Distribute material
    - Specify scope and priorities
    - Introduce the product (Author)
- **Individual checking –** *responsible: Inspectors*
    - Exam the product individually and report all defects to leader
        - 2 – 3 pages per hour or 100-200 lines of code standard rate
- **Inspection meeting –** *Leader, Recorder, Reader and Inspector*

    - Inspect, produce a defect list
    - Review list and make an exit decision:
        - Accept with no further verification
        - Accept with rework verification
        - Reinspect – redo the process
- **Edit and follow-up**
    - Author resolves items
    - Inspection leader verifies that all items are closed
- **Collected data**
    - General inspection data
    - Classification – e.g. logic problem, sensor problem
    - Categories – e.g. missing, extra, incorrect
    - Ranking – e.g. catastrophic, marginal, negligible

Other software reviews:

- **Management reviews** – Check deviations from plans, performed by management.
- **Technical reviews –** Evaluate conformance to specifications and standards, performed by technical leadership with a high volume of materials.
- **Walk-though –** An informal atmosphere where the Author presents, leads and controls the discussion.
- **Audit –** External evaluation of conformance to specification and standards.

Software metrics

The practice of software metrics is to measure the previously mentioned quality factors. We do this via metrics, i.e. combinations of measurements; measurements can consist of:

- No. of pages in a *document*
- No. of elements in a *design*
- No. of lines of *code*
- Iteration length of the *process*
- Avg. no. of hours to learn a system to decide *quality*

Measurements on Quality Factors

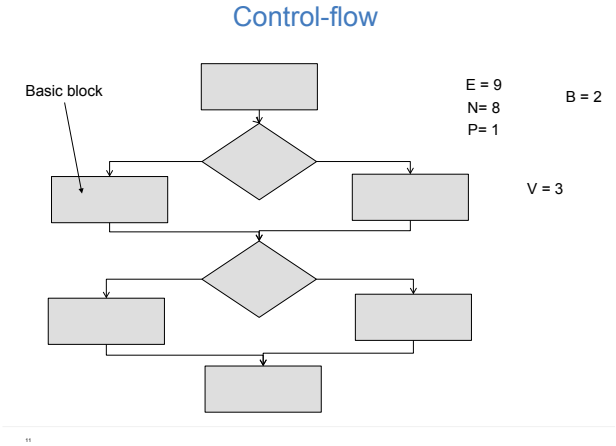
We often calculate reliability with Mean Time To Failure (MTTF).
Reliability = MTTF/(1+MTTF) (Estimated probability)
Or **Failure intensity = (1-R)/t** (More straight-forward measurement)

Similar patterns are**:**
Availability = MTTF/(MTTF+MTTR) where MTTR = Mean Time To Repair
Maintainability = 1/(1+MTTR)

Cyclomatic complexity

The Cyclomatic complexity V(G) of a flow graph G is calculated: **V(G) = E - N - 2P,** where:

- E = Num. of edges
- N = Num. of nodes
- P = Num. of disconnected parts in the graph



*A control-flow graph with cyclomatic complexity V(G) = 9 - 8 - 2*1 = 3. This can represent e.g. two if-statements.*

A number of specific metrics also exist:

- **Usage based metrics -** Example
  - Description: Number of good and bad features recalled by users.
  - Obtain data: Set up a test scenario. Let test users run the scenario. Collect number of good and bad features in a questionnaire afterwards.
  - Calculate the metric: Take the average of number of good and bad features.
  - Relevant quality factor: **Relevance** – many good and few bad features indicates a good match with the users' mind-set.
- **Verification and validation metrics** – Example
  - Description: Rate of severe defects found in inspection of design description.
  - Obtain data: Perform an inspection according to your process. Make sure that severity is in the classification scheme.
  - Calculate the metric: Divide the number of defects classified with highest severity with total number of defects in the Inspection record.
  - Relevant quality factor: **Safety** – a high proportion of severe defects in design indicates fundamental problems with the solution and/or competence.
- **Volume metrics** – Example
  - Description: Number on non-commented lines of code.
  - Obtain data: Count non-commented lines of the code with a tool.
  - Calculate the metric: See above.
  - Relevant quality factor: **Reliability** – it is often hard to understand a large portion of code; the fault density is often higher for large modules.
- **Structural metrics** – Example
  - Description: Maximum depth of inheritance tree.
  - Obtain data: Count the depth of the inheritance tree for all classes.
  - Calculate the metric: Take the maximum value of the classes.
  - Relevant quality factor: **Understandability** – It is hard to determine how a change in a higher class will affect inherited/overridden methods.

- **Effort metrics** – Example
    - Description: Time spent in testing.
    - Obtain data: Make sure that testing activities are distinguished in time reporting forms. Make sure that all project activities are reported.
    - Calculate the metric: Sum the number of hours for all activities in testing for all people involved.
    - Relevant quality factor: **Testability** – a comparably long testing time indicates low testability.

Software Quality Assurance (SQA)

Quality can both be something in the eyes of the beholder, something we learn to recognize or value-based on the market where we often can measure it objectively. A number of levels of quality assurance exists:

- **Appraisal –** Detection
- **Assurance –** Prediction
- **Control –** Adjusting
- **Improvement –** Reduce variation, increase precision

Usability engineering

Throughout the iterative process we increase the maturation and knowledge of the project group and therefore decrease the risk, this is done by evaluating goals of:

- Relevance
- Effiency
- Attitude
- Learnability

Mature organization

A mature organization has a higher level of work accomplished and well defined roles where they do things well, not necessarily good.

Capability Maturity Model Integration (CMMI)

CMMI is used to decide the level of maturation of an organization, the model have five stages, each with a number of process areas to focus on.

1. Initial – Over-commited, no repetition of success.
2. Repeatable – Process adherence is evaluated and we can repeat a previous success.
3. Defined – Tailored from own standards, improved processes with detailed descriptions. Originally the minimum level.
4. Managed – Frequent measures and statistics is kept for goals, products and processes. Have a high predictive capability.
5. Optimising – Everyone committed to continuous improvement and an innovative climate is paired with evaluation of new technology. Performance gaps are known.

Example ***process areas*** of CMMI:
Requirements Management (REQM) – Level 2 (Requirements)
Purpose: Manage requirements of the project's products and product components and to ensure alignment between those requirements and the project's plans and work products.

Requirements Development (RD) – Level 3 (Requirements)
Purpose: Elicit, analyze, and establish customer, product, and product component requirements.

Technical Solution (TS) – Level 3 (Design)
Purpose: Select design and implement solutions to requirements. Solutions, designs, and implementations encompass products, product components, and product related lifecycle processes either singly or in combination as appropriate.

Project Planning (PP) – Level 2 (Project Management)
Purpose: Establish and maintain plans that define project activities.

Risk Management (RSKM) – Level 3 (Project Management)
Purpose: Identify potential problems before they occur so that risk handling activities can be planned and invoked as needed across the life of the product or project to mitigate adverse impacts on achieving objectives.

Process and Product Quality Assurance (PPQA) – Level 3 (SQA)
Purpose: Provide staff and management with objective insight into processes and associated work products.

Organizational Process Definition (OPD) – Level 3 (Process)
Purpose: Establish and maintain a usable set of organizational process assets, work environment standards, and rules and guidelines for teams.

Configuration Management (CM) – Level 2 (Process)
Purpose: Establish and maintain the integrity of work products using configuration identification, configuration control, configuration status accounting, and configuration audits.

Verification (VER) – Level 3 (Testing)
Purpose: Establish and maintain the integrity of work products using configuration identification, configuration control, configuration status accounting, and configuration audits.

Validation (VAL) – Level 3 (Testing)
Purpose: Demonstrate that a product or product component fulfills its intended use when placed in its intended environment.

ISO 9000-3

Is a guideline from the International Organization for Standardization which is built on the principles of:

- Customer focus
- Leadership
- Involvement of people
- Process approach
- System approach to management
- Continual improvement
- Factual approach to decision-making
- Mutually beneficial supplier relationships

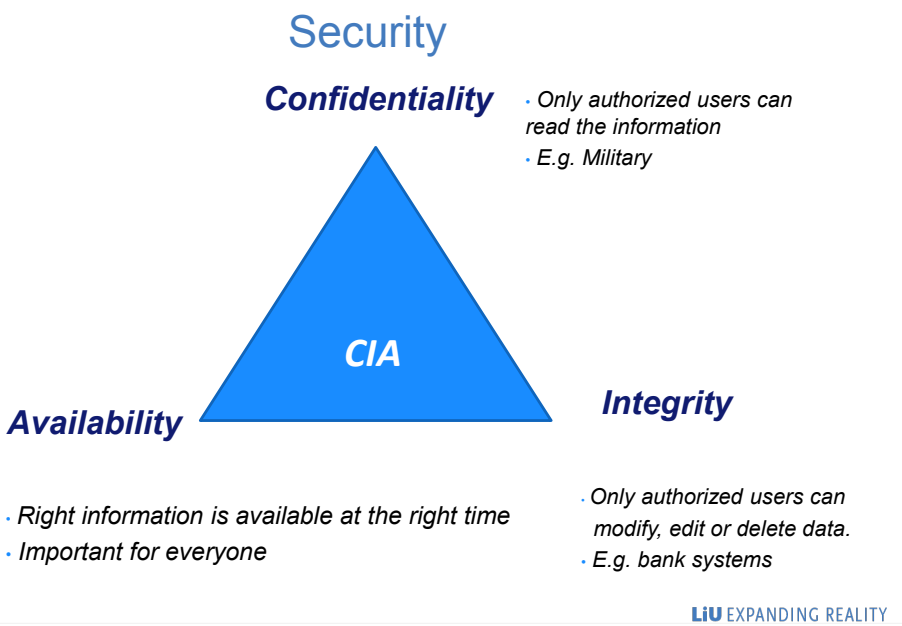
Total Quality Management (TQM)

is a set of management practices throughout the organization, geared to ensure the organization consistently meets or exceeds customer requirements. TQM places strong focus on process measurement and controls as means of continuous improvement.

Has 7 main principles:

1. Quality can and must be managed
2. Processes, not people, are the problem
3. Don't treat symptoms, look for the cure
4. Every employee is responsible for quality
5. Quality must be measurable
6. Quality improvements must be continuous
7. Quality is a long-term investment

Software security



CIA – Confidentiality, Availability and Integrity is central for software security.