**POM 01 – Introduction**

*Introduction:*

Project management is a **problem solving activity**

* **Analysis**: Understand the problem nature and break the problem into pieces
* **Synthesis**: Put the pieces together into a large structure that prepares for the solution

*Summary of Polya‘s book:*

* **First**: understand the problem
* **Second**: devise a plan
* **Third**: carry out your plan
* **Fourth**: examine the solution obtained [untersuche erhaltende Lösung]
* **Methodologies**: Collection of techniques, heuristics and tools uniﬁed by a philosophical approach [vereinheitlicht durch einen philosophischen Ansatz]
* **Techniques**: Formal procedures for producing results using some well-deﬁned notation
* **Heuristics**: Informal collection of steps
* **Tools**: Instruments or automated systems that help in accomplishing a technique or supporting heuristics

*Software project management: (traditional definition)*

Project Management is a collection of techniques, methodologies, tools and heuristics that support the development of

* a **high quality software** system
* with a given **budget**
* before a given **deadline**
* **while change occurs** (bei **modern** hinzugefügt)

*Methodology:*

Provides guidance, general principles and strategies for selecting methods and tools

[welche Methode für eine bestimmte Art der Anwendung geeignet ist]

Key questions - criteria:

* **costumer** (how much interaction with costumer?)
* **planning** (how much planning in advance?)
* **reuse** (how much reusing past solutions?)
* **modeling** (how much modeled before it is coded?)
* **process** (how much detail should the process be defined?)
* **project monitoring** (how often the work controlled and monitored?)

*Software project: workﬂows*

* Decision Making and Issue Tracking -> JIRA
* Meeting Management, Documentation, Collaboration -> Confluence
* Continuous Integration -> Bamboo
* Conﬁguration Management -> Bitbucket
* Continuous Delivery -> HockeyApp

**POM 02 – Project Organization**

*Stages of Team Development:*

* Stage 1: **Forming** (goal alignment [Zielausrichtung], get to know each other)
* Stage 2: **Storming** (member forms opinions of each other, ideas compete for consideration)
* Stage 3: **Norming** (resolving conflict, roles are accepted)
* Stage 4: **Performing** (empowering behaviours, most effective stage, trust)

*Difference between group and team:*

* **Group**: number of people that have some relationship to one another
* **Team**: Any group of people involved in the same activity with a common goal

**Project Management Terminology**

*Project Deﬁnition:*

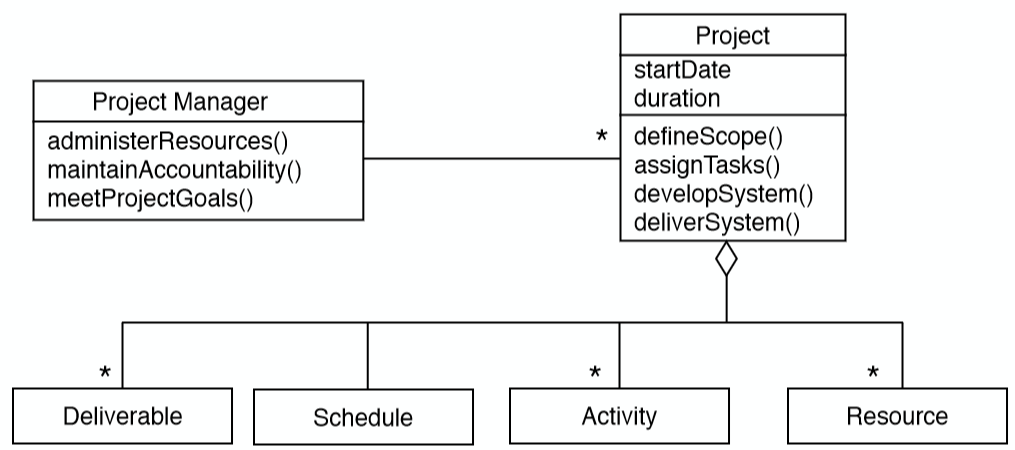
A project is an undertaking, limited in time, with a clear goal and a speciﬁc budget, requiring a concerted effort

A project is managed by a **project manager** who:

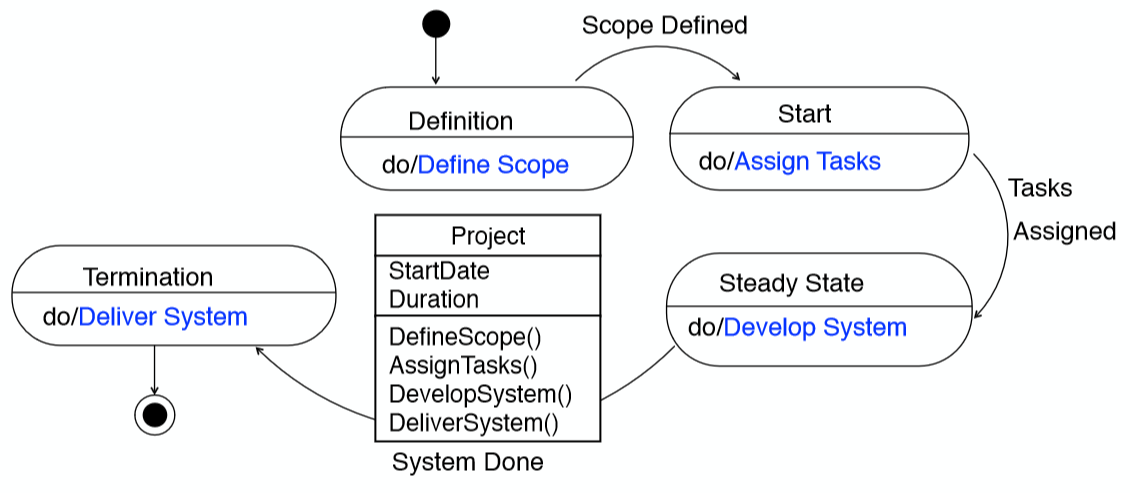
* Administers the resources
* Maintains accountability [Rechenschaftspflicht]
* Makes sure the project goals are met

**Post mortem analysis and retrospectives help to improve yourself in future projects**

*Reﬁned Object Model:*



*Dynamic Model:*



*Role:*

A role deﬁnes a set of responsibilities, a responsibility is a duty or task a person is required to do. Examples of roles and corresponding responsibilities:

* **Project manager** (administer the resources, make sure the project goals are met)
* **Analyst** (analyse the application domain, create a taxonomy of the domain abstraction)
* **System architect** (decompose the system into subsystems)
* **Tester** (design and implement tests)
* Developer (Analyst, System architect), Liaison (Tester)

*Assignments of Roles to Participants:*

* **One-to-One** (ideal but rare)
* **Many-to-Few** (each project member assumes several “hats”, danger of over-commitment)
* **Many-to-“Too-Many”** (some people don’t have significant roles, lack of accountability)

*Bad Role Assignments:*

* **Incompetence** (wrong person fills the wrong role)
  + **Peter Principle**
  + “Employees who perform their roles in a hierarchy with competence are promoted to a higher level until they reach a level where they are no longer competent. There they remain forever” 🡪 Employees are incompetence at their level
* **Useless roles** (role exists only to minimize damage control)
  + **Dilbert’s Law**
  + “Companies tend to systematically promote their least-competent employee to management (generally middle management), in order to limit the amount of damage they are capable of”
* **Increase of Bureaucracy** (role swells unnecessarily, simply because it can)
  + **Parkinson’s Law** [maximale Zeit wird ausgenutzt 🡪 nicht früher fertig als geplant]
  + „Work expands to ﬁll the time available for its completion“ 2 reasons:
    - 1) “Ofﬁcials want to multiply subordinates, not rivals”
    - 2) ”Ofﬁcials make work for each other”

*Key concepts for mapping roles to people:*

* **Authority**: make decisions between people and roles
* **Responsibility**: commitment [Verpflichtung] of a role to achieve speciﬁc results
* **Accountability**: tracking a task performance to a speciﬁc person
* **Delegation**: Binding a responsibility assigned to one person (including yourself) to another person

*Delegation:*

Three reasons for delegation:

1. **Time Management**: To free yourself up for other tasks
2. **Expertise**: The most qualiﬁed person makes the decision
3. **Training**: To develop another person’s ability to handle additional assignments

* **You can delegate authority, but you cannot delegate responsibility**
* **You can only share responsibility**

*Task:*

A task describes the smallest amount of work monitored (tracked) by the project manager

* Typically less than 2-4 working days effort
* If a task size is too large, the task should be renamed into an activity
* an activity is decomposed into smaller tasks that allow monitoring

*Activities:*

* An activity is a major unit of work
* Activities allow to separate concerns [trennen von Anliegen]
* Precedence relations often exist among activities [Vorrangbeziehungen]

**Examples:**

* Planning, Requirement Elicitation, Analysis, System Design, Configuration Mgmt., …

*Unit of Work:*

* Activities are often grouped again into **higher-level** activities
* **Unit of Work:** A task or an activity that contains other tasks and lower-level activities

*Project Function:*

An activity that spans the entire duration of a software project

**Examples:**

* project management, documentation, testing, continuous integration, release management;

*Work Package:*

* A **task** or **activity** is speciﬁed by a **work package** which contains:
  + Description of work to be done
  + Preconditions for starting, duration, required resources
  + Work products to be produced, acceptance criteria for it
  + Risks involved

*Work Product:*

A **work product** is the visible outcome of a unit of work (model, presentation, piece of code)

Work products that have to be given to the customer are called **deliverables**



**POM Class 3**

**Functional Organization:**

* Grouped into departments (finance, production, sales, analysis, design, testing…)
* Every department addresses an activity (function)

**Key** **Properties**:

* Projects are pipelined through the departments (from research, then development..)
* Different departments address identical needs (IT infrastructure, Configuration mgt.)
* Only few participants are completely involved in a single project

**Advantage:**

* Members have a good understanding of their area

**Disadvantages:**

* High chance for duplication of work

**Project-based organization**

People are assigned to a project, each of which has a problem to be solved in a certain time within a given budget

**Key** **Properties**:

* Teams are assembled when a project is created
* Every project has a project manager
* A participant is involved in a single project **only**

**Advantages**:

* Responsive to new requirements (the project can be tailored around the problem)
* New people can be hired who are familiar with the problem
* No idle time for project members (Wartezeit)

**Disadvantages**:

* Difficult to assemble a team
* Roles and responsibilities need to be defined at the beginning

**Flat vs. Gradual staffing**

**Gradual staffing**: Ramped up by hiring people as needed. It is motivated by saving resources at the early parts of the project.

**Flat staffing**: All participants are assigned at the start of a project. Taken from a pool of available people.

**Matrix Organization**

People from different departments of a functional organization are assigned to work on one or more projects (less then 100% of their time in one project)

**Advantages**:

* Teams for projects can be assembled rapidly from the departments
* Expertise can be applied to different projects as needed
* Consistent reporting and decision procedures can be used for projects of the same type

**Disadvantages**:

* Team members are often not familiar with each other
* Team members have different working styles
* Team members must get used to each other

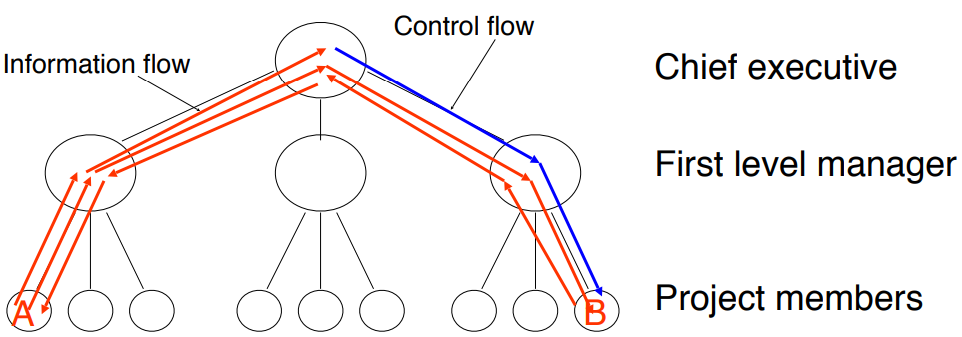
**Challenges:**

* Team members working on multiple projects have competing demands for their time
* Multiple work procedures and reporting systems are used by different team members
* “**Double-boss problem**”: two different bosses with different focus
* Department and project interests might be in conflict with each other

**Project organization structures**

* **Decision structure**: Models the control flow: Who decides what?
* **Reporting structure**: Who reports their status to whom?
* **Communication structure**: Models the information flow: Who facilitates communication with whom?

**Line organization problem:**



* Information flow in a hierarchical project organization does not work well with unexpected changes
* The manager is not necessarily always right and might even misunderstand communication requests
* Improving information flow through non-hierarchical project organizations
  + Cutting down on bureaucracy (direct communication is possible)
  + Reduces development time
  + Decisions are expected to be made at each level
  + Hard to manage (who is in control in case of conflicts?)

**Communication skills**

A software project manager as well as a software engineer needs to acquire several **skills**:

* **Collaboration**: negotiate requirements with the client and with members from your team
* **Presentation**: present a major part of the system during a review
* **Technical writing**: write part of the proposal, part of the project documentation
* **Management**: facilitate a team meeting, find compromises
* **Communication** is critical for the success

**Communication event vs. mechanism**

**Communication event**: information exchange with defined objectives & scope

* **Scheduled**: planned communication
* **Unscheduled**: event-driven communication. Examples: request for change
* **Examples**: Problem Definition, Project Review, Client Review;

**Communication mechanism**: tool or procedure that can be used to deal with a communication event

* **Synchronous**: same time
* **Asynchronous**: different time

**🡪 Communication Event is supported by Communication Mechanism**

**Communication mechanisms**

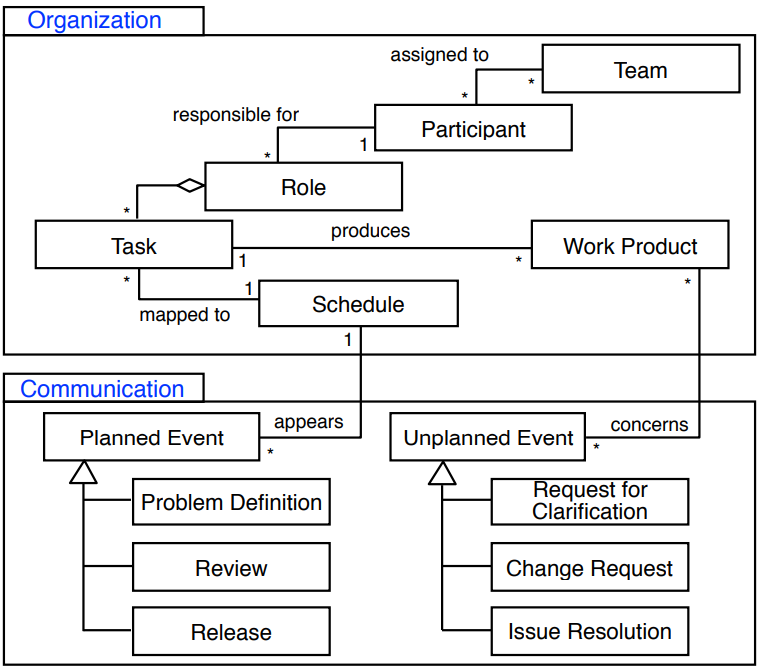
Synchronous examples:

* **Informal Meeting**:
  + Unplanned conversation
  + + Cheap and effective for resolving simple problems
  + - Information loss, misunderstandings
* **Formal Meeting**
  + Planned conversation, project reviews…
  + + Effective for resolving issues
  + - High costs

Asynchronous examples:

* Email, Chats, Wikis

UML View of project management basics



**Meeting management**

* plan a productive meeting
* **constant level of quality** and a **structured process**
* provides **templates, roles** and **guidelines**

**Meetings**

* A team should meet at least once a week
* Use a **structured agenda**
* Share important information (**meeting minutes**)
* Every participant is aware of his/her **role** in the meeting

Meeting Roles:

**Primary Facilitator:**

* Organizes the meeting, creates the agenda, guides the meeting as moderator

**Minute Taker:**

* Creates meeting minutes (Protokoll) during the meeting and hand them out

**Time Keeper**:

* Keeps track of the time allowed to interrupt any person in the meeting

**Post mortem analysis (project retrospective)**

* Empirical study method, get knowledge about past projects
* Performed after most important milestones or at the end of the project
* reveals problems and solutions more frequently

**Software Project Management Plan (SPMP)**

* controlling and planning document
* specifies technical and managerial approaches [Ansätze]
* other documents:
  + requirements analysis document (RAD)
  + system design document (SDD)
* changes in one of these 3 documents imply changes in the other 2 documents

**Project Agreement**: A document written for a client that defines:

* Scope, duration, cost and deliverables for the project
* Exact items, quantities, delivery dates, delivery location

The form of a project agreement can be a contract, a statement of work, a business plan, or a project charter

**POM 04**

**The IEEE 1058 standard**

* describes structure of a software project management plan, applicable to any type or size
* specifies the format and contents of software project management plans

**Two styles of navigation [Gladwin 1964]**

1) European navigation (Traditional Planning)

2) Polynesian navigation (Agile Planning)

➡Main difference: reaction to unexpected events (“change”)

**Properties of software project plans**

* Useful at the beginning of a project
* Useful also for projects if the outcome is predictable or when no major change occurs

**Software lifecycle**

A software lifecycle is similar to humans lifecycle.

* **Software lifecycle** (also software process): Set of activities and their relationships to each other to support the development of a software system
* **Software lifecycle model** (also software process model): An abstraction that represents a software lifecycle for the purpose of understanding, monitoring, or controlling the development of a software system.

**Where do we need models?**

* **Communication**: The model provides a common vocabulary. An informal model is often used to communicate an idea
* **Analysis/Design**: Models enable developers to reason about the future system
* **Archival**: Compact representation for storing the design and rationale of an existing system (Documentation)

**Support communication** (communication models)

* Most often used in the **early phases** of a project and during informal communication
* The model is used only to communicate an **idea to a person**
* Communication media: whiteboard, a mockup, or even a napkin [Grundriss]

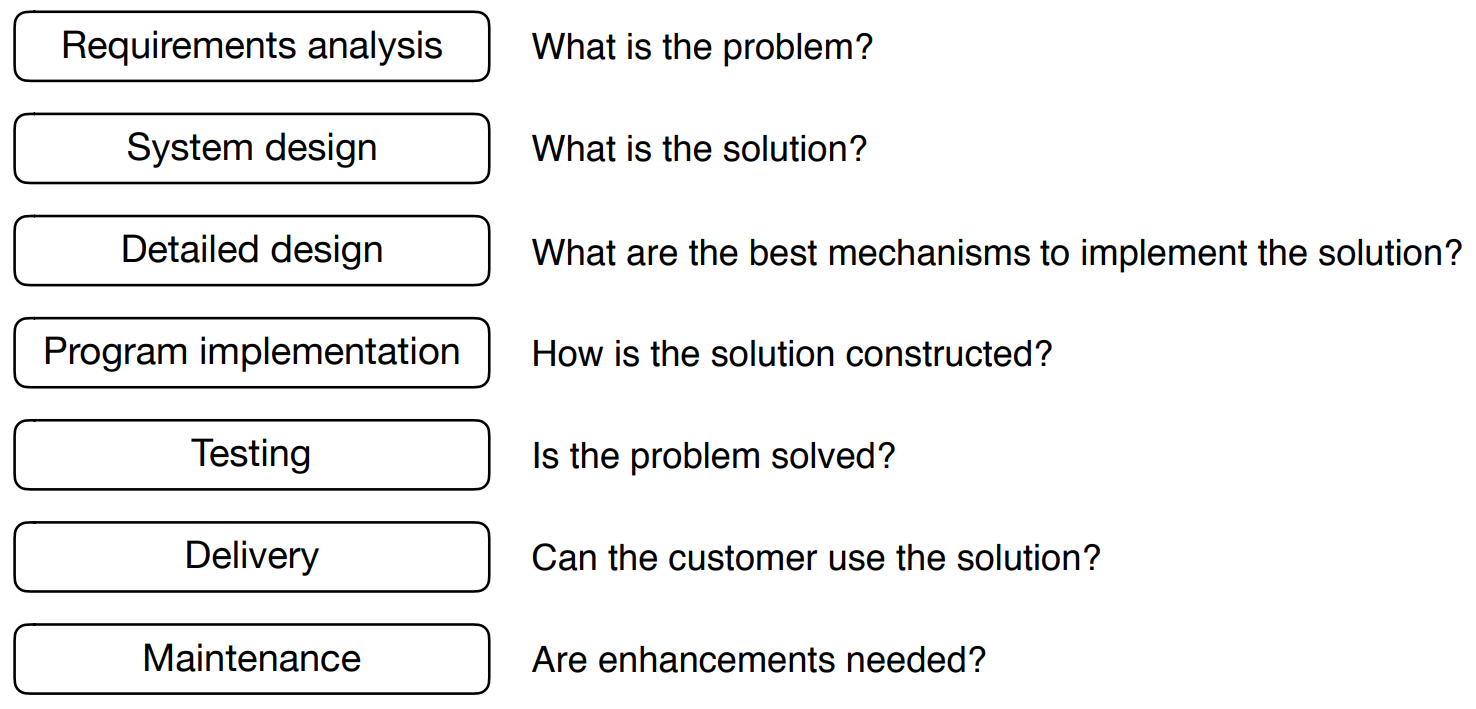
**Support analysis and design** (specification model)

* The model provides a representation that enables developers to reason about the system
* The model is used to communicate an **idea to a tool**
* UML is our preferred notation for models that support analysis and design

**Managerial challenges of modeling**

* Formalizing knowledge is expensive
  + Takes time and effort from developers
  + Requires validation and consensus
* Models introduce redundancy
  + If several models depict (abbilden) the same aspects of the system, all of them must be updated
* Models can become too complex
  + As the model complexity becomes similar to the complexity of the system, the benefit of having a model is reduced significantly

**Software development activities (Example)**



**Tailoring**

* There is no “one size fits all” software lifecycle model that works for all software projects
* **Tailoring**: adjusting a lifecycle model to fit a project
* **Naming**: adjusting the naming of activities
* **Cutting**: removing activities not needed in the project
* **Ordering**: defining the order the activities take place in

**Modeling a software lifecycle (POM 04 ab Folie 37)**

* **Functional** **model** of a software lifecycle
  + Scenarios, user stories, Use case model
* **Structural** **model** of a software lifecycle
  + Object identification, Class diagrams
* **Dynamic** **model** of a software lifecycle
  + Sequence diagrams, state chart and activity diagrams

**Overview of software lifecycle models**

* Sequential
  + Waterfall Model, V-Model
* Iterative
  + Spiral Model, V-Model XT, Unified Process
* Agile
  + Extreme Programming (XP), Kanban, Scrum

**Summary**

* A software lifecycle model has the purpose to understand, monitor, or control the development of a software system
* A software lifecycle model consists of activities and their relationship
* We use UML activity diagrams to model software lifecycles

**POM 05 – SCRUM I**

*Controlling software development with a process:*

How do we control software development? Two opinions:

1. Through **organizational maturity** (Humphrey 1989)
2. Through **agility** (Schwaber 2001):

*Defined vs. empirical process:*

* **Defined process:**
  + Planned, follows strict rules
* **Empirical process:**
  + Not entirely planned inspect and adapt

*Defined process control model:*

* Requires that every piece of work is completely understood
  + **Deviations are seen as errors that need to be corrected**
* Given a well-defined set of inputs, the same outputs are generated every time

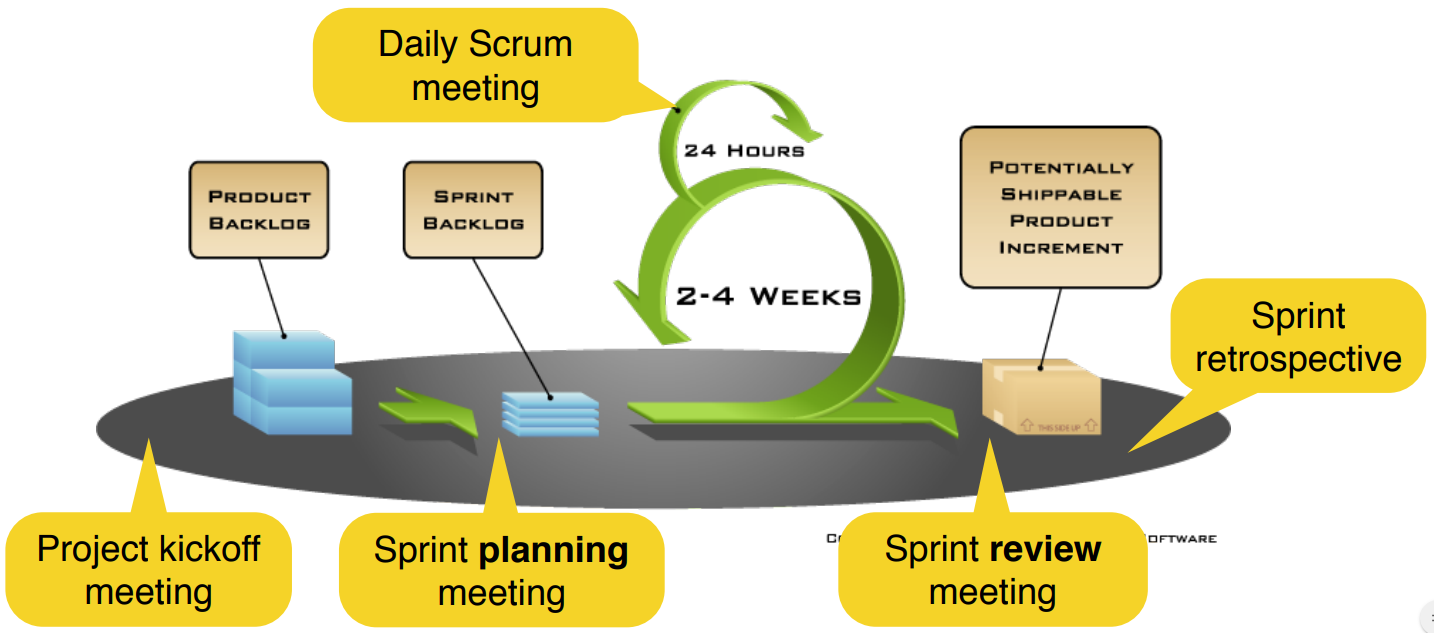
*Empirical process control model:*

* Imperfectly defined process, not all pieces of work are completely understood
  + **Deviations, errors and failures are seen as opportunities that need to be investigated**
* **Expects the unexpected**: control and risk management is exercised through frequent inspection

*3 Scrum artifacts:*

1. **Product backlog**: List of requirements for the whole product
2. **Sprint backlog**: List of requirements and tasks for one iteration (“sprint”)
3. **Potentially shippable product increment**: Release to the product owner that contains all results of the current sprint

*Scrum meetings:*



*5 Scrum meetings:*

1. **Project kickoff meeting** (start of the project): Create and prioritize the product backlog
2. **Sprint planning meeting** (start of each sprint): Create the sprint backlog
3. **Daily scrum meeting** (every day, 15min): Share status, impediments and promises (in a standup meeting)
4. **Sprint review meeting** (end of each sprint): Demonstrate the realized backlog items to the product owner (and other stakeholders)
5. **Sprint retrospective** (after the sprint): Inspect the previous sprint and create a plan for improvements to be enacted during the next sprint (sometimes combined with the sprint review meeting)

*Scrum team with 3 roles:*

1. **Product** **owner** (defines the product, responsible for results)
2. **Scrum** **master** (resolves impediments, responsible for the process)
3. **Developer** (Development team – Self-organizing and cross-functional, realizes the product increment)

*Sprint:*

* creates a potentially shippable product increment
* Typically 2-4 weeks long
* Starts with **sprint planning meeting**
  + Create the Sprint Backlog: Selection of items to be implemented in the Sprint
  + Important: Development Team and Product Owner select the items together
* Ends with **sprint review meeting**
  + Release and deliver the application (product increment)
  + Important: Product Owner gives feedback
* Sprint review meeting and sprint planning meetings are typically combined into a single meeting with the product owner
* The Scrum team can additionally perform a sprint retrospective meeting

*Product backlog:*

* Collection of items (typically requirements, e.g. user stories, scenarios, etc.) prioritized by the product owner
* The product backlog can always be changed and reprioritized during the projects
* Created on the basis of the problem statement during the project kickoff meeting or in the phase before the actual project starts

*Priority:*

* priority describes the importance of the requirement for the software
* Examples for a priority scheme (that we also use in JIRA)
  + **Prio 1 = Critical** (Candidates for the first development sprints, must be part of the first product increment)
  + **Prio 2 = Major** (Must be realized within the project, can be realized in one of the following product increments)
  + **Prio 3 = Minor** (Desirable, if there is enough time)
  + **Prio 4 = Not Important** (Might not be realized at all)
* Priorities of requirements can change during the project in Scrum
* **Prioritization is done by the product owner**

*Estimation with level of difficulty:*

* Estimation is not easy for developers —> in particular beginners struggle with it
* **Simplified estimation**:
  + Small (**S**)
  + Medium (**M**)
  + Large (**L**)
  + Extra Large (**XL**) —> too big, split it into smaller issues!
* **More sophisticated estimation**: In the sprint planning meeting, the team decides on the difficulty of the backlog items by estimating story points (1/2, 1, 2, 3, 5, 8, 13, 21, 100) using e.g.: Poker planning

**POM 6**

**Requirements in Scrum**

All Requirements are collected in the product backlog

The product owner can be seen as a requirements engineer:

* Elicit, define and prioritize requirements

**Types** of requirements:

* User stories, scenarios, use cases
* Functional and non functional requirements

**User stories**

A user story includes a sentence that describes what the user does or needs

* **Example: As <role>, I can <feature> so that <reason>**

As **sales representative** I can **determine the scale of discount of a customer** so that **I am able to tell him a concrete offer with the correct price.**

**Properties of a good user story: INVEST**

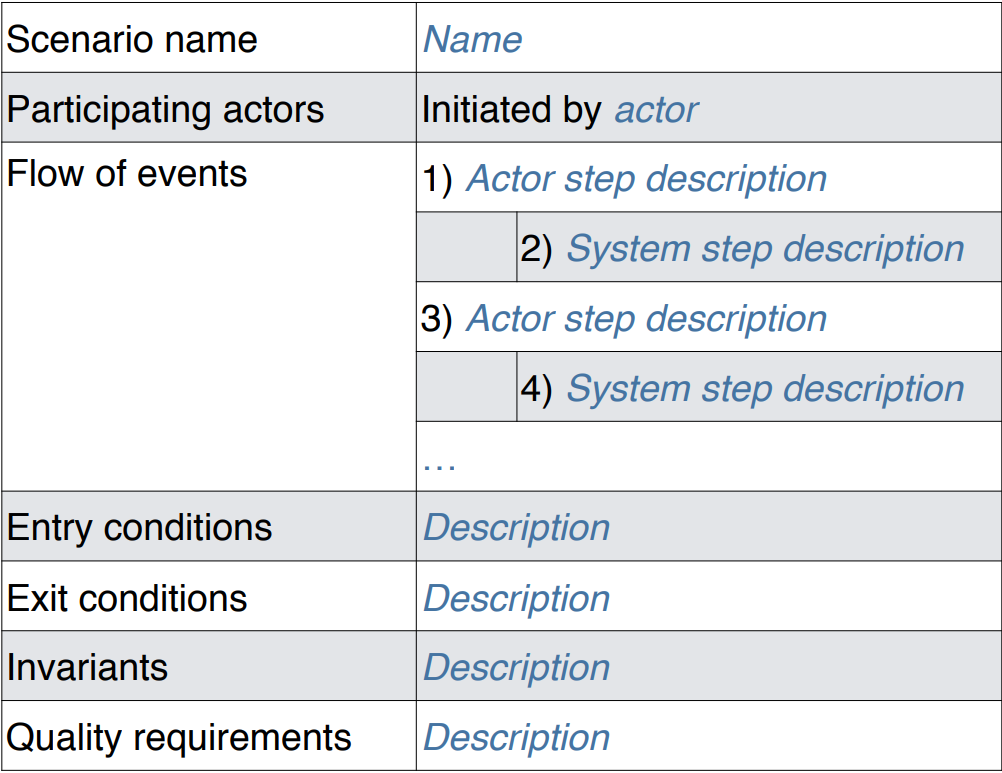
* **Independent** - avoid overlapping user stories
* **Negotiable** - a user story is not a contract, but a basis for discussion between development team and product owner
* **Valuable** - for the user and the business. And **Vertical**: plan and develop features, not layers
* **Estimable** - the stories in the product backlog represent the basis of the project plan
* **Small** - too large user stories must be partitioned into smaller ones to avoid an over-proportional increase of complexity
* **Testable** - if a user story is not testable, it might not be of real value for the product. This also implies realizability

**Acceptance criteria**

* Conditions that a software product must satisfy to be accepted by a user, customer or other stakeholder
* Set of statements, each with a clear pass/fail result
* either a criterion is met or it is not
* Acceptance criteria are typically written on the back of the user story

**Scenarios**

Instance of a use case. A scenario represents a concrete sequence of interactions between one or more actors and the system.



**Relation between user stories, use cases and scenarios**

* All focus on functional requirements, but also reference non-functional requirements
* **Scenarios** are typically created **during analysis**
* Use cases and scenarios typically cover a larger scope and are more formal than user stories
* **User stories** are more informal and typically created **during requirements elicitation**

**JIRA supports Scrum**

* Create and manage the product backlog
  + Create backlog items (e.g. scenario, user story)
  + Estimate the difficulty
  + Define sub-tasks for the unit of work during development
  + Prioritize backlog items and tasks
* Create and manage sprint backlogs
  + Plan sprints
  + Track progress in active sprints
  + View reports about finished sprints

**Sprint planning meeting**

* Development team estimates the difficulty for the items in the product backlog
* Development team and product owner select product backlog items that can be realized
* Development team negotiates with the product owner how many items it can realize
* The product owner defines when an item is accepted (e.g. using acceptance criteria)
* Important: The sprint backlog cannot be changed by the product owner within the Sprint to protect the team from too many changes

**Daily scrum meeting**

* Main purpose: Risk reduction by early information sharing and discussion
* 15 min **standup meeting** every day
* Every developer answers the following 3 questions:
  + **Status**: What did you do since the last meeting?
  + **Impediments**: Are there any impediments in your way? (also called blockers)
  + **Promises**: What do you promise to resolve until the next meeting?

**Sprint review meeting**

* The development team delivers a product increment including the realized items from the sprint backlog
  + Send it before the review meeting to the product owner
* The development team demonstrates the product increment to the product owner
* The product owner or other stakeholders provide feedback and decide whether the items are realized completely
* Unrealized items move back to the product backlog
* The product owner can add new requirements to the product backlog or change existing
* The review meeting can include a sprint retrospective (can also be a separate meeting)

**Potentially shippable product increment**

* Each sprint focuses on the incremental creation of a working system
* The product increment can be thrown away or delivered ➡The product owner decides

**Sprint retrospective meeting**

* Scrum master and development team meet to discuss how the previous sprint worked out ➡ The product owner can participate if necessary
* There are different retrospective techniques ➡ Often it is most effective to brainstorm about things that worked well / did not work well
* Each team member is asked to identify specific things that the team should:
  + Start, stop or continue doing

**Scrum as methodology @a**

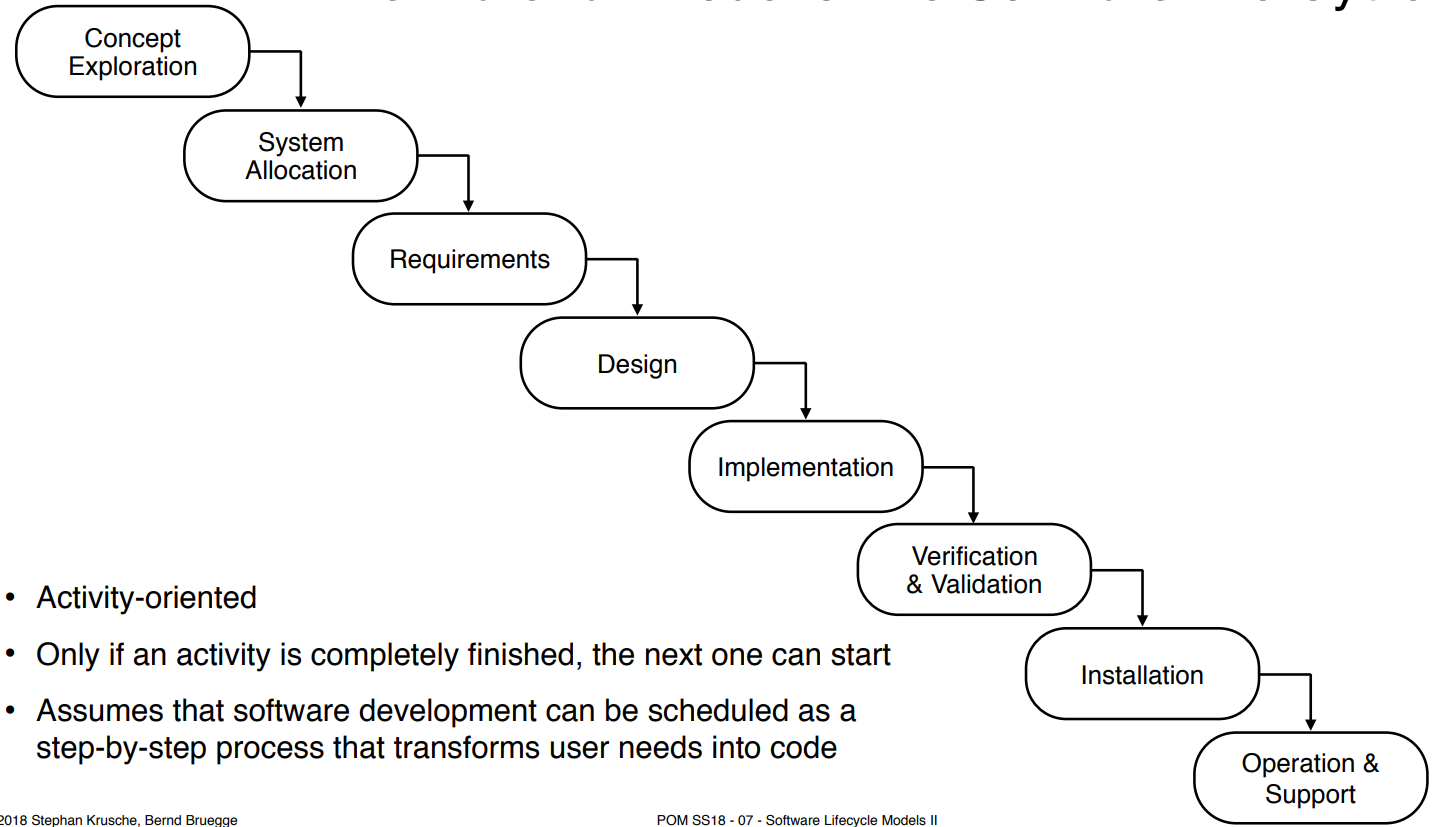
* **Involvement** **of** **the** **customer**: Onsite customer (“co-located”, product owner)
* **Planning**: Checklists and incremental daily plans (“Daily Scrum”)
* **Reuse**: Checklists from previous projects
* **Modeling**: Models may or may not be used
* **Process**: Iterative, incremental and adaptive process
* **Control and Monitoring**: Risk management distributed across daily meetings

**Summary**

* Scrum does not define which type of requirements should be used
* Most Scrum teams prefer user stories
* JIRA supports Scrum artifacts and activities
* Create and manage the Product Backlog and its backlog items (e.g. User Stories / Scenarios)
* Plan the Sprint by choosing backlog items and by adding Sub-Tasks
* Track the status of a Sprint using a Taskboard and the progress of a Sprint with a Burn Down Chart
* Prioritize and estimate backlog items —> more about estimation in a later exercise
* Review the Sprint using acceptance criteria

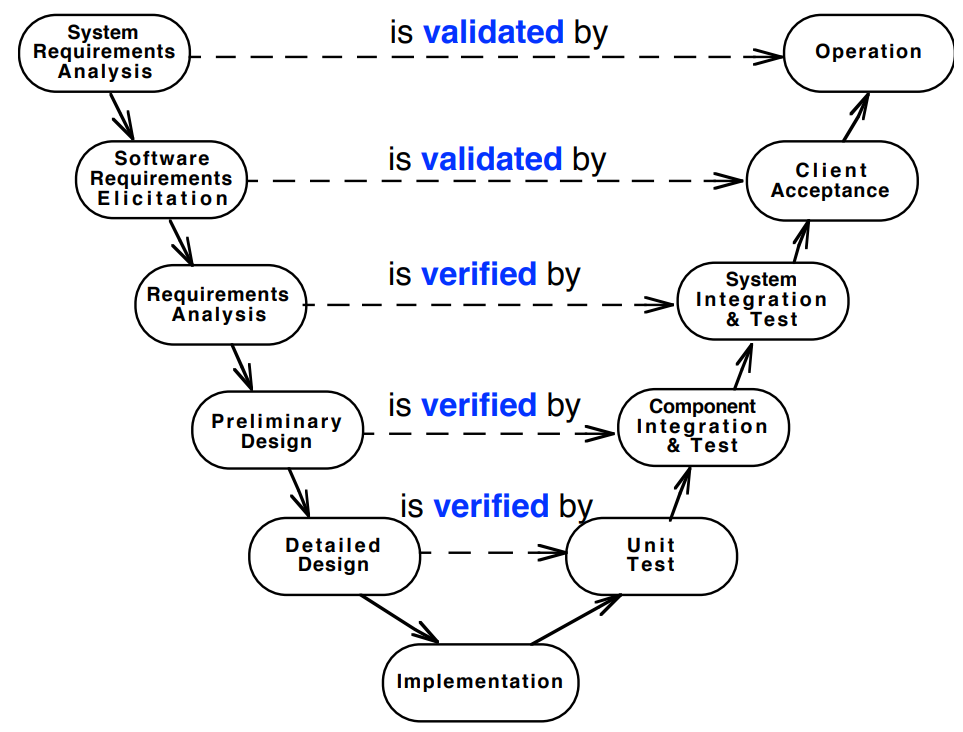
**POM7**

**The Waterfall Model of the Software Life Cycle**



* The model uses a prescribed sequential execution of activities
* The software development can be scheduled as a linear step by step process
* The goal is to never turn back once an activity is completed
* The key feature of the model is the **verification** activity at the end of each activity that ensures that the activity does not introduce unwanted or deletes mandatory requirements

**Activity Diagram of the V Model**



* The horizontal object flow describes the information flow between activities of same abstraction level

**Validation vs. Verification**

Validation**:**

* Assurance that a product meets the needs of the customer
* **Informally: "Are you building the right thing?"**

Verification:

* Evaluation whether a product complies [entspricht] with a regulation, requirement, specification or imposed condition or not
* **Informally: "Are you building it right?"**

**Properties of sequential models (Waterfall and V Model)**

* Nice milestones
* No need to look back (linear system)
* Always one activity at a time
* Easy to check progress during development, e.g. 90% coded, 20% tested
* But software development is non-linear!

**Definition: iterative vs. incremental**

* **Iterative** means “re-do” or “re-work”

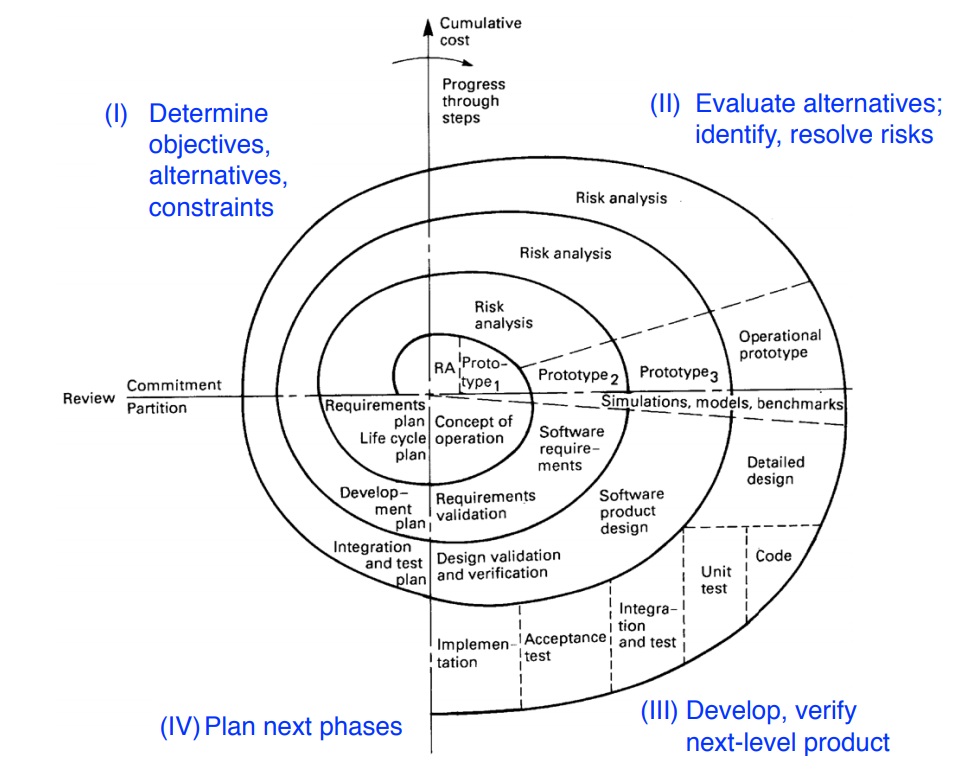
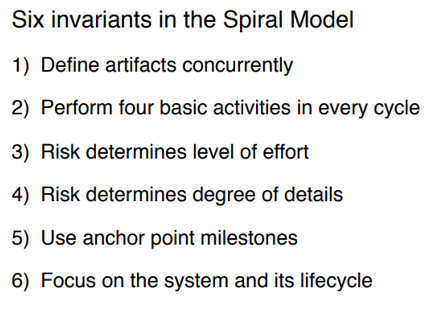
➡ Iterative development helps you to improve your product

* **Incremental** means “to add onto something”

➡ Incremental development helps you improve your process

**Spiral model**

* 9 iterations (cycles)
* 4 basic activities that must be applied in each iteration

**3 types of prototypes**

1. Illustrative prototype

* Develop the user interface with a set of storyboards
* Implement them on a napkin or with a user interface builder
* Good for a first dialog with the client

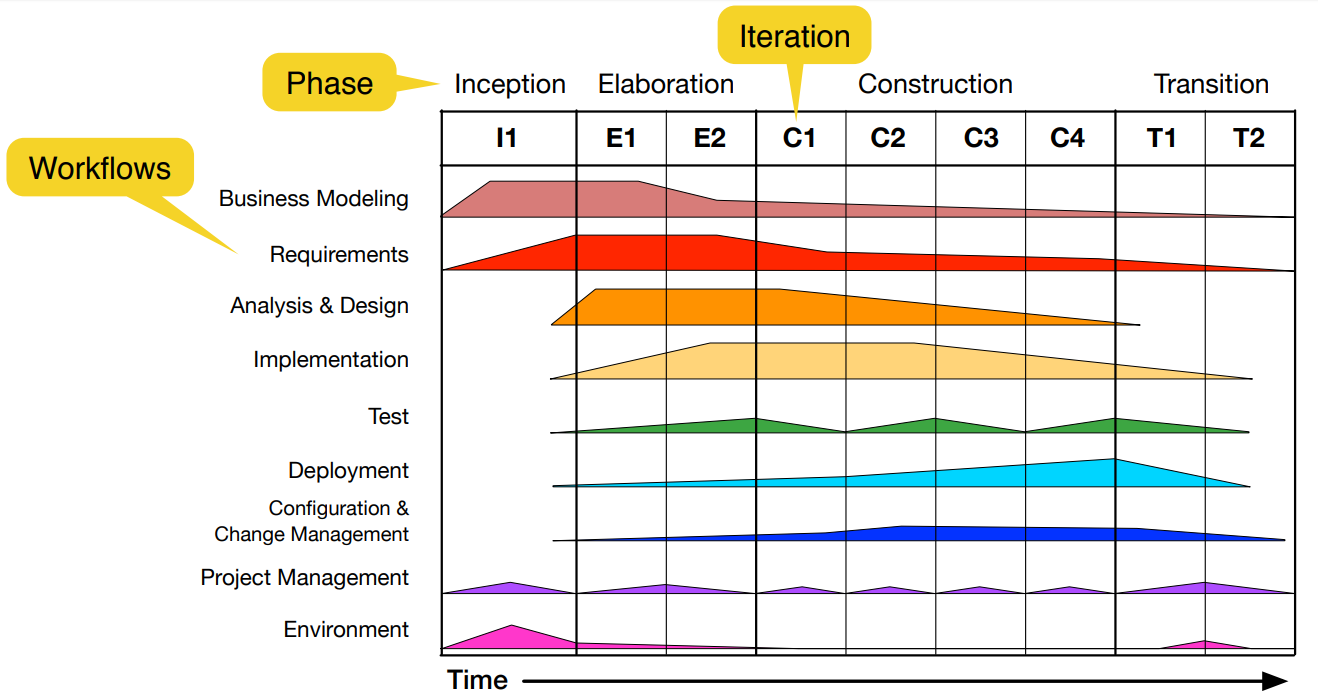
1. Functional prototype

* Implement and deliver an operational system with minimum functionality
* Then add more functionality
* Good for incremental development

1. Exploratory Prototype ("Hack")
   * Implement part of the system to learn more about the requirements

**Vertical vs. horizontal prototype**

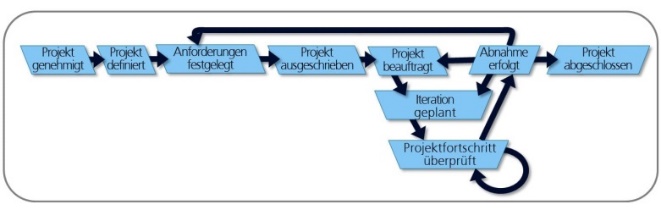


**Unified process**

* An iterative and incremental lifecycle model built on the idea of **cycles** in the lifetime of a software system
* Each cycle consists of 4 **phases**: inception, elaboration, construction, transition
* Each phase can be iterated. During the duration of an iteration, several **workflows** are performed in parallel

**V-Model XT**

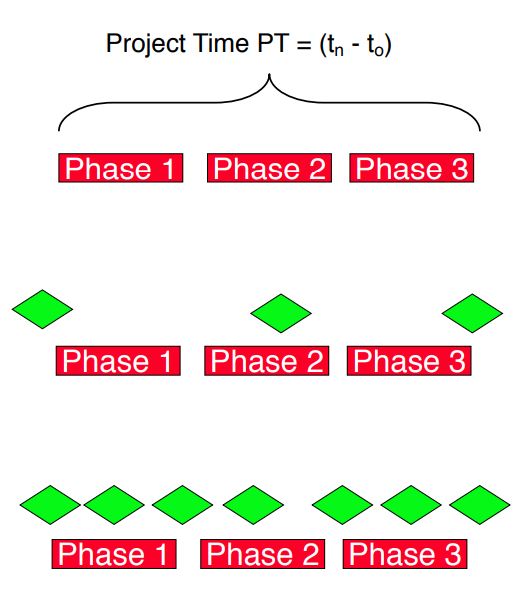
* The V-Model XT is the successor of the V-Model, XT = eXtreme Tailoring
* Goals:
  + Minimization of project risk
  + Improvement and guarantee of quality
  + Reduction of cost
  + Improvement of communication
* Focus on products instead of activities
  + No defined chronology of activities
* V-Model XT describes who has to do what and when in a project
* Each type has a project execution strategy
  + Each strategy is roughly itself a process model, consisting of milestones and decision points



**V-Model XT Tailoring**

* Tailoring means selecting the process components to be used in the project
* Static tailoring occurs at project definition time (before the project actually starts)
* Dynamic tailoring may occur during project execution

**Frequency of change and choice of software lifecycle model**

PT = Project Time

Time MTBC = Mean Time Between Change

* **Change rarely occurs** (MTBC >> PT)
  + **Sequential** model: Waterfall Model, V-Model
  + Open issues are closed before moving to next phase
* **Change occurs sometimes** (MTBC ≈ PT)
  + **Iterative** model: Spiral Model, Unified Process, V-Model XT
  + Change occurring during phase may lead to the iteration of a previous phase or cancellation of the project
* **Change is frequent** (MTBC << PT)
  + **Agile** model: Scrum, Kanban, Extreme Programming (XP)
  + Change during a phase can lead to reengineering the requirements or the design

**Extreme programming (XP)**

* Main goals
  + Avoid over-planning
  + Improve software quality
  + Improve responsiveness to changing customer requirements
* **Terminology**: iteration, deliverable, release
* 5 fundamental principles: rapid feedback, assume simplicity, incremental change, embracing change, quality work
* 4 roles: developer, customer, manager, coach
* 12 practices: how to approach the development process

**XP practice: test driven development (TDD)**

* XP is based on test-driven development (TDD)
* Do not code before you have tests
* Also called **Test first**
* Tests are not only for testing functional requirements, they are also for testing nonfunctional requirements and adherence to standards
* Write only tests that could possibly fail

**XP practice: refactoring**

* Refactoring: A change that leaves system behavior unchanged, but enhances simplicity, flexibility, understandability, and/or performance
* Keep all tests running after refactoring

**XP practice: pair programming**

* + All production code is written with two people looking at one machine
  + There are two roles in each pair:
  1. Driver - Thinks tactically: worries about the implementation, writes the tests and the production code
  2. Navigator (“Observer”) - Thinks strategically: Is this whole approach going to work? What test cases may fail? Can we simplify the system to make this problem go away?

**XP terminology: iteration**

* An iteration implements one or more user stories
  + Iterations are “time-boxed”
  + Iteration finishes always on a fixed date
  + Typical duration of an iteration: 2 to 4 weeks
* Number of implemented features is variable
* (Jeder Iteration alle 4 Bereiche abdecken)

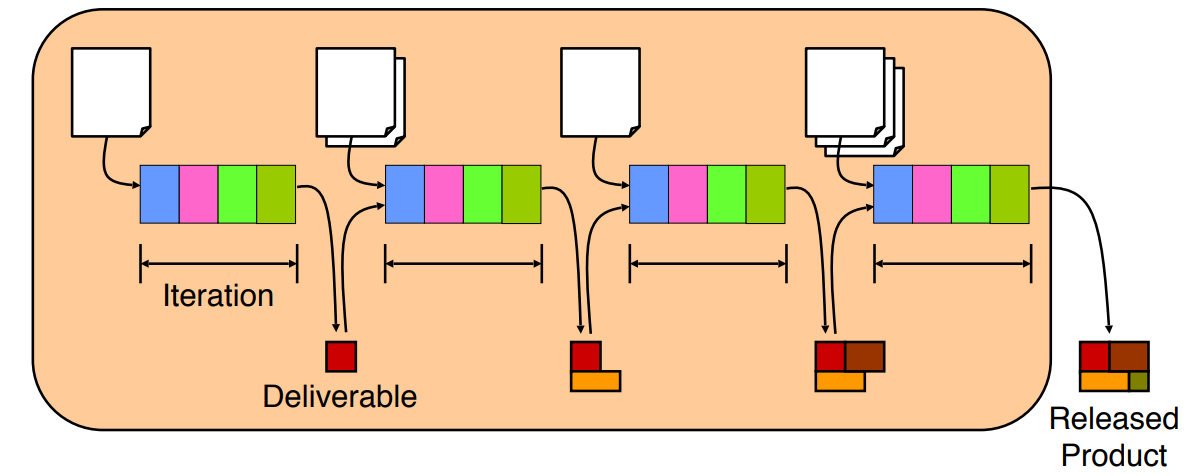


**XP terminology: deliverable**

* Deliverable: result of an iteration
  + Coded, tested, and potentially shippable
* Small addition of functionality

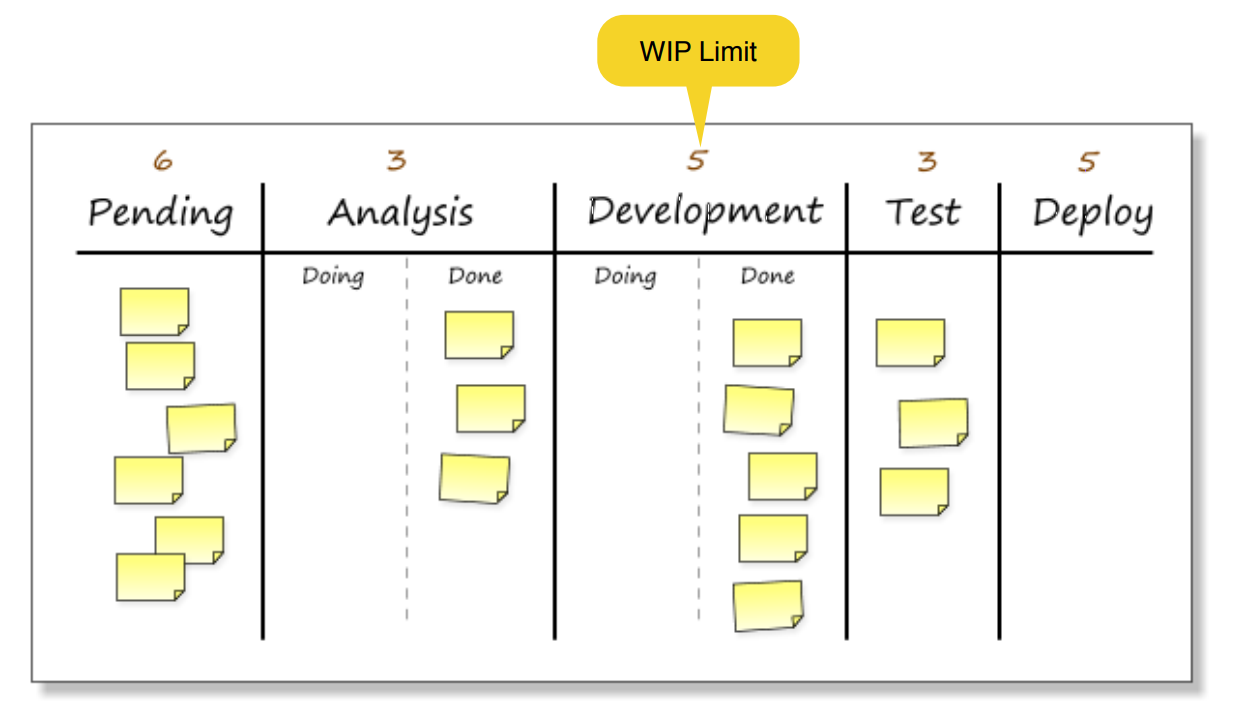
**XP terminology: release**

* A release consists of several iterations that add related functionality
  + Usually every 2 to 6 months
* When should you release?
  + When the customer sees a big improvement over the last release

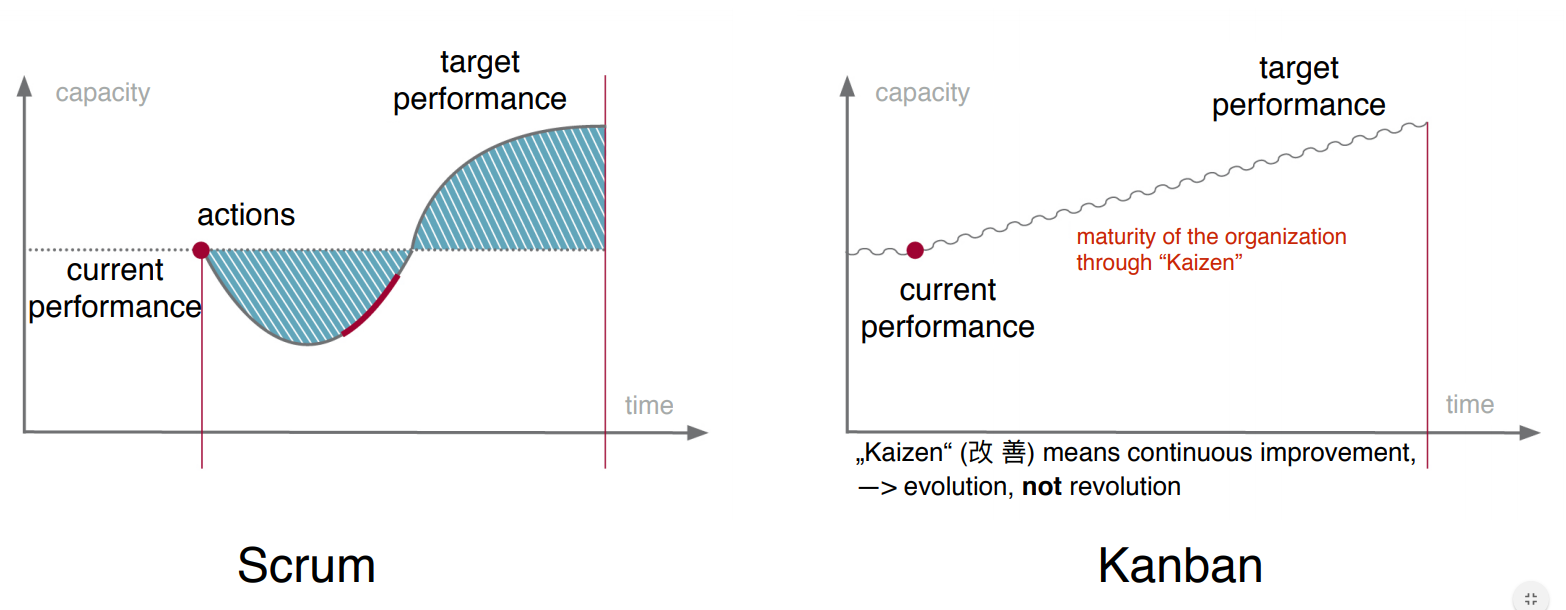


**Kanban**

* Stands for transparency, optimization and personal responsibility
* Does not solve problems, it simply makes them visible!
* Begins with the current state and process of a system



**Scrum is revolutionary, Kanban is evolutionary**



**Situations where agile methods might not be appropriate**

* When it is not supported by the company culture
* Project too big for regular complete integration
* Where it inherently takes a long time to get feedback
* Where you can’t realistically test

**POM class 8 – Contracting**

Questions to be answered in Contracting:

* Expected deliverables
* What is the schedule
* Price
* Responsibilities of the contractor and the client
* What about change management

Important:

* Contract defines the basics of a project (requirements, money, time)
* Defines basic rules of collaboration
* How to deal with uncertainties (change, innovation)

There are always two projects (Client Project, Contractor Project)

**Client** **Project** - Get a system

**Contractor** **Project** - Build a system

**Contracting in the Project Life Cycle: Roles**

**Client** (also: Customer, Purchaser, Contracting Entity)

* Define initial requirements
* Request proposals
* Monitor proposal projects
* Acceptance testing

**Contractor** (also: Supplier)

* Submit offers (bidding)
* Ship results (software -> concrete deliverable, services)

**internal** **project** **without** **any contract**…

* Department A orders a software, department B develops the software •
* Who is the client? Who is the contractor?
* How do they define the agreement?

**Influence factors** in contracting:

* Goals
* Business Case
* Initial estimations
* Legal restrictions

**What’s in a contract?**

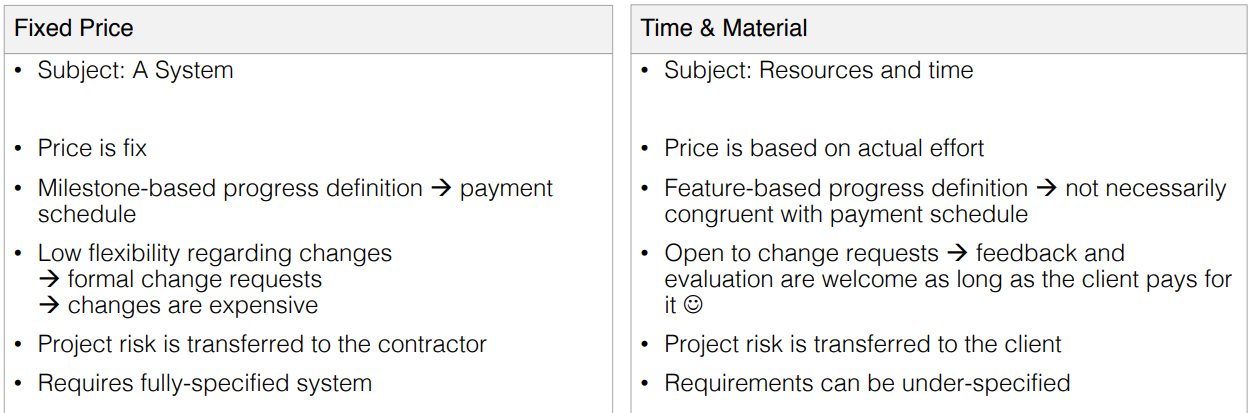
* Services and responsibilities (Price, Deadline, functional req.)
* Warranty and liability
* Provisions
* Copyrights
* Regulations and standards, compliance
* A project usually starts with a dialog (acquisition)…
* Acquisition strategies:
  + Pro-active
  + Re-active
  + Client request/request for an offer
  + Call for bids/submission
* A contract is always concluded based on the accepted offer

Acceptance procedure should define

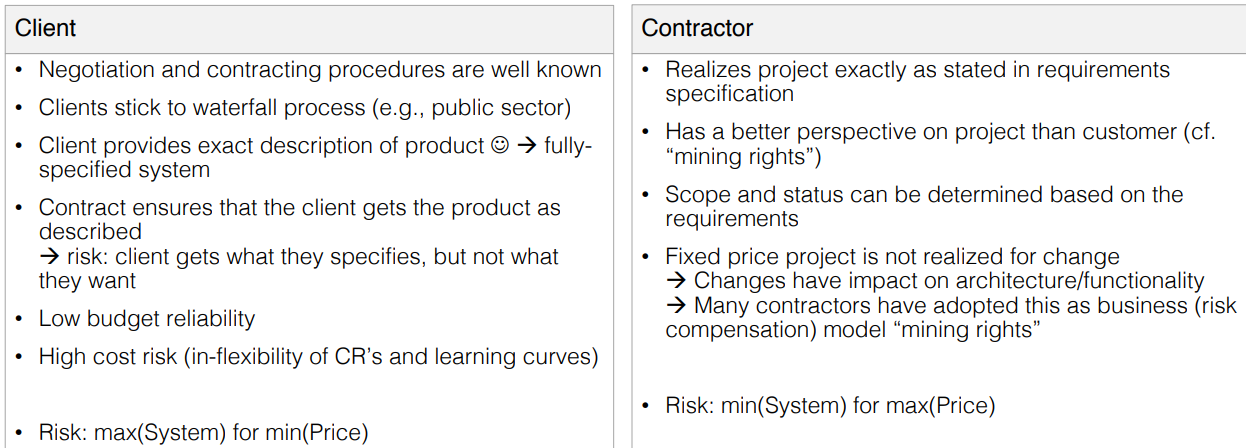
**Contractor**: Shipping

**Client**: Acceptance test

Contract Types



Fixed



Time & Material



Contracts should be designed according to the “**win-win”** interaction strategy (fair contract)

**Why is it necessary to have some alternatives**

* Companies mix processes
* For different reasons
  + Dependable systems and agile development practices
  + Dealing with requirements; keep flexibility
  + Connect host organization and team
* Consequence: The way contracting is implemented changes as well

**Practices**

Several practices exist to make contracts more flexible and to adopt agile methods

* Maximum price
* Change for free
* Exit point/Money for nothing

**Maximum Price**

Idea: work according to time & material approach, but set a limit

* Define and estimate the system
* Define the max. price for the system
* Define the cost per unit (e.g., story point)

**Change for Free**

**Idea**: client is participating in the project all the time

* Gives continuously feedback
* Has learning curves (better understands the system)
* Allow the client to add features, but
* Identify other features that have be removed/stalled
* Update the contract accordingly

Example: Given n features for a Sprint. Client decides to add a new feature x to the next Sprint. As compensation, a feature y is removed. Removal of y is mentioned in an updated contract

**Exit Points/Money for Nothing**

**Idea**: define an approach in which a project can be stopped, but good performance is awarded

* Define exit points (when and under which conditions to stop a project)
* Define payments to “compensate” or award early termination

**Example** (**exit** **point**): Given a project has 6 planned sprints

* Exit point: initial phase = 2 sprints for n story points
* Contractor is paid for 2 sprints -> if no exit: whole project budget
* If exit, contractor can demand compensation

**Example** (**money** **for** **nothing**): Given a project is finished early

* Contractor can be awarded with extra money, e.g., 10k for every month saved

**Combined Practices**

Idea: combine fixed price and time & material, and other practices into a defined, but flexible contract

Basic instrument: fixed price contract (maximum price)

Work is done and paid according to time & material

Exit points are defined (incl. cash flow, exit conditions)

Money for nothing is implemented to award good performance

**How to handle failure?**

Contract-relevant failure in a project can occur for many reasons

It is important to be prepared, for this:

* Define **Warranty**: when do liability and damage compensation start?
* Define **Ownership**: who owns the project results?
* Define **Copyright**: who is authorized to publish and use project results?
* Define **Escalation**: don’t go to court immediately; give mediation a chance

**Contracting: Important to know**

Contracts define the basic rules of collaboration – they set the stage…

They **define:**

* Functionality/deliverables and their respective quality
* Responsibilities
* Time/Schedule
* Money

Two **basic** types:

* Fixed price
* Time & material

Several **additional** **practices** exist to adopt flexibility and to reflect actual software development business (changing requirements, technology, etc.)

Contracts **define** **the** **rules** -> It’s not the goal to outrun the partner

Contracts **need** a **balance** (keyword: risk share)

Contracts **need** **to** **be** **fair** -> all participating parties have rights and duties

**POM 09 – Usability Management**

*Terminology:*

**Usability**: measures how well a user can utilize the system functionality and is measured by five categories

1. **Learnability**: how easy/fast can a user learn the use of the system?
2. **Efficiency**: how many steps does a user require to accomplish a particular task?
3. **Memorability**: how quickly can a user reestablish proficiency?
4. **Errors**: how many errors do users make, how severe are these errors, and how easily can they recover from the errors?
5. **Satisfaction**: how pleasant is the design of the user interface?

"**The system is easy to use**": one of the most frequently misused terms, especially in advertising (often these systems are actually unusable)

"**Unusability**": the user has extreme difficulties to learn or to use the system

*User interfaces are hard to design:*

* The developer and the user are not the same person
  + Software engineers communicate mostly with other developers
  + User interface development is about communicating with users
* The user is always right …
  + Consistent problems are the system’s fault
* … but the user is not always right
  + Users are not designers
* User interface takes a lot of software development effort
  + ~50% of design, implementation and maintenance
* Managers must be involved (usability management)

**How to address usability: prototyping:**

*Prototyping definition:*

* is externalizing and making concrete a design idea for the purpose of evaluation.
* is an early sample or model built to test a concept or process or to act as a thing to be replicated or learned from.

*Why Prototyping?*

* Instant gratification (sofortige Zufriedenstellung)
* Tangibility (Greifbarkeit): a prototype helps to understand a system early on
* Improves poor communication
* Allows early decision making
* Mistakes can be found early: **“We want instant prototypes. They allow us to make more mistakes faster”**

*Prototyping techniques:*

* Haptic prototypes
* Paper prototypes
* Wireframe prototypes (also called mockups)
  + Term from 3D computer graphics
  + Used for the schematic presentation of a screen or webpage
* Storyboard
  + Sequence of Wireframes
* Low-fidelity and high-fidelity prototypes
* Scenario based video prototypes

**User interface prototyping**

*Tips for creating paper prototypes:*

* Create as many prototypes as possible
* Invest as little time as possible for the first iterations, they don‘t have to be perfect and you get feedback anyway!
* Ask persons for feedback who are in the target group of your software
* Don‘t limit yourself by thinking „Is this even possible?“
* Involve as many persons as possible

*Failures are helpful:*

Henry Petrovski:

* Better information comes from designs that fail rather than from those that succeed
* Reason: failures draw more scrutiny; without failure, complacency sets in
* „Success in engineering is defined by its failures“

*Knowledge must be falsifiable:*

* Karl Popper (“objective knowledge”):
  + There is no absolute truth when trying to understand reality
  + One can only build theories, that are “true” until somebody finds a counter example
* The truth of a theory is never certain
  + We can only use phrases like: “by our best judgment”, “using state of the art knowledge”
* **Falsification**: the act of disproving a theory or hypothesis

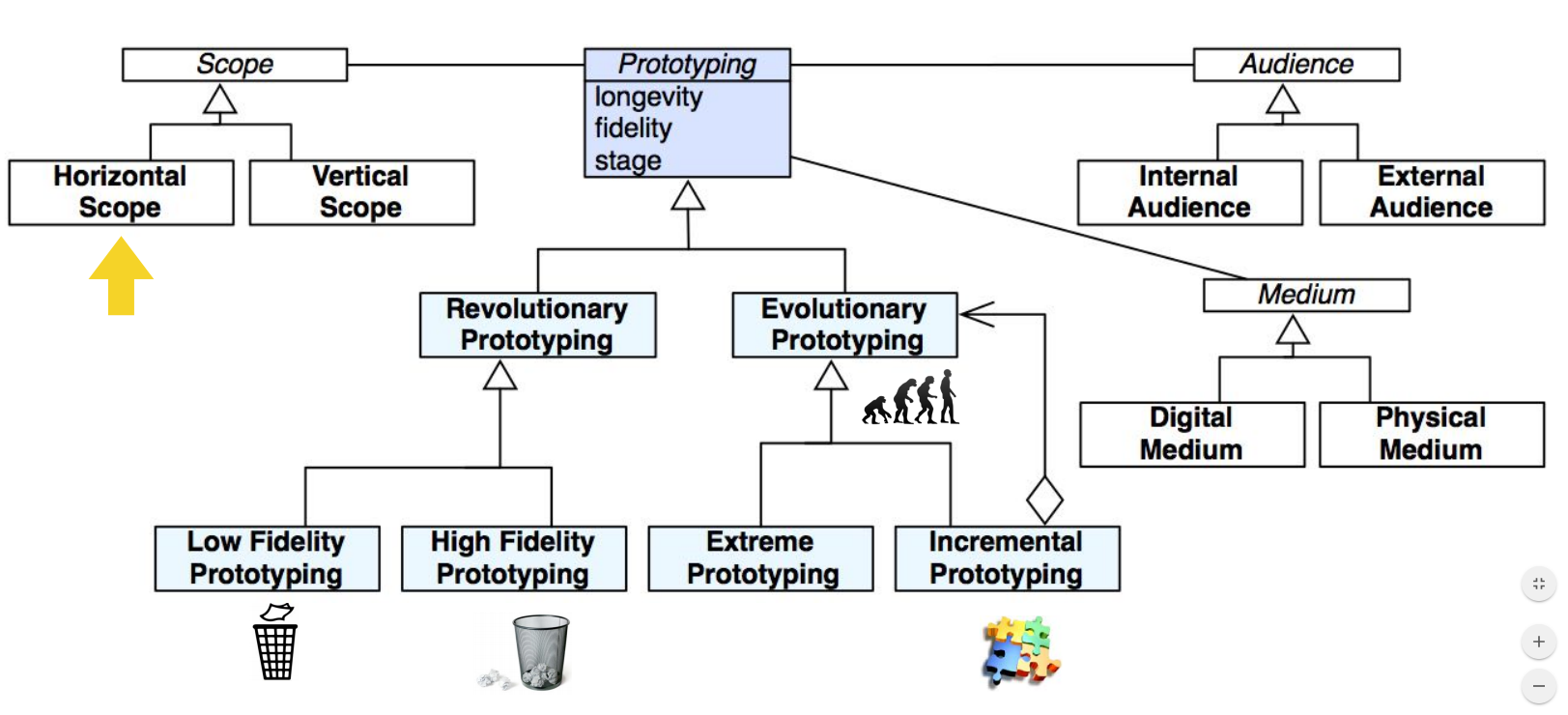
*Consequence for software systems:*

* In software engineering, any system, including a user interface, is a model and thus a theory:
  + We build models to find counter examples
  + Techniques: requirements validation, user interface testing, review of the design, source code testing, system testing, etc.
* **Testing**: the act of disproving a model
* **Usability testing**: the act of testing a user interface, i.e. disproving a user interface model
* We can do these tests with prototypes

*A typical prototyping process:*

1. Low-fidelity prototype
2. High-fidelity prototype
3. Application

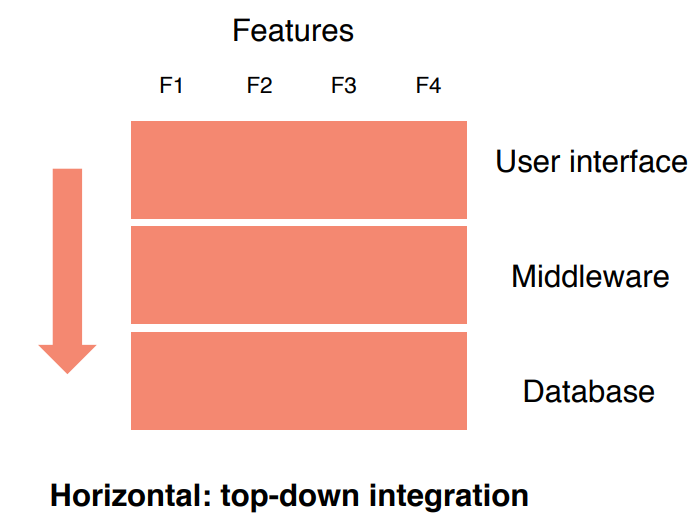
*Types of Prototyping:*



*Horizontal vs. vertical prototypes:*

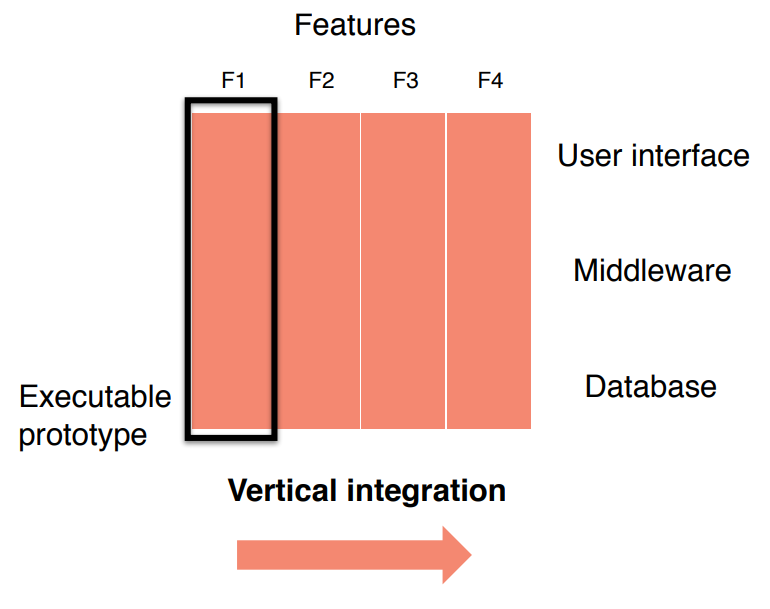
**Horizontal prototypes**

* Show wide range of features
* Horizontal integration
* Bottom up, top down
* Used in linear processes:
* No full implementation up to the end

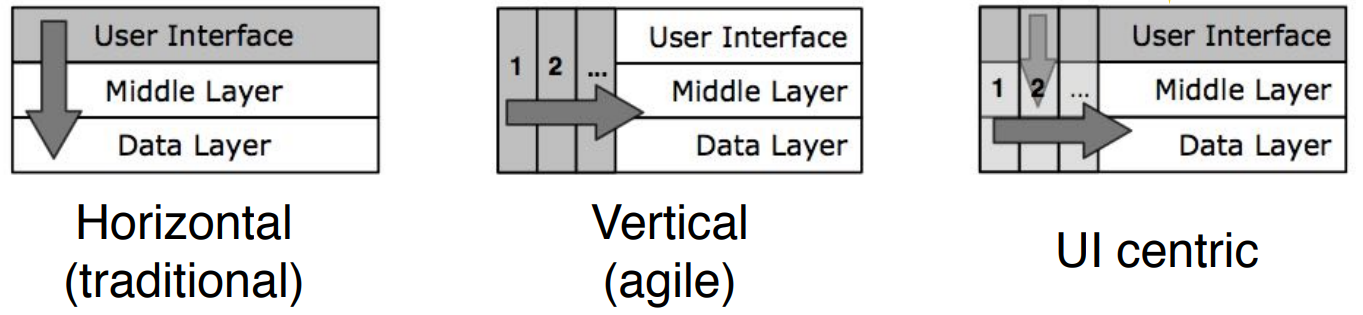


**Vertical prototypes**

* Show small range of features (e.g. scenario, user story)
* Full implementation of these features
* Vertical integration
* Used in agile processes



*Integration approaches used during prototyping:*



*Methods to reach good usability:*

* **Usability testing**: Watching a user interact with the user interface of the system
  + Usability testing uses scenario-based design
  + Involves the creation of a test scenario
  + The user performs a list of tasks while the observer watches and takes notes, and compares the observed with the specified/expected behaviour
* **Heuristic evaluation**: A usability engineering method to find usability problems in a user interface design
* The difference between usability testing and heuristic evaluation is similar to the difference between walkthrough and inspection (review techniques)

*Nielsen’s 10 heuristics:*

* **Meet expectations**

1. Match the real world
2. Consistency & standards
3. Help & documentation

* **The user is the boss**

1. User control & freedom
2. Visibility of system status
3. Flexibility & efficiency

* **Handle errors**

1. Error prevention
2. Recognition, not recall
3. Error reporting, diagnosis, and recovery

* **Keep it simple**

1. Aesthetic & minimalist design

*Heuristic evaluation:*

* An application of Nielsen’s 10 heuristics
* An inspection method
* Performed by usability experts
* Basic steps

1. An evaluator inspects the user interface thoroughly
2. Compares the user interface against the 10 heuristics
3. Provides a list of usability problems

*Good heuristic evaluation:*

* Justify every problem with a heuristic
  + You cannot just say “I don’t like the colors”
  + Better: “Too many choices on the home page violates Nr. 10 Aesthetic & Minim.”
* List every problem
  + If an interface element has multiple problems, list them all
* Go through the interface at least twice
* Don’t limit yourself to the 10 heuristics
* Nielsen’s heuristics are a good start to compare against

*Evaluating prototypes:*

* Heuristic evaluation can already be used for the evaluation of prototypes
  + It should start early in the development process
  + Do not wait for the final product
* Heuristic evaluation works on
  + Sketches
  + Paper prototypes
  + Early software prototypes
* “Missing element” problems are harder to find on sketches
  + Because you’re not actually using the interface, you aren’t blocked by feature’s absence
  + Look harder for them

*Review: Revolutionary vs. evolutionary prototyping:*

* **Revolutionary prototyping**: get experience with a throwaway prototype
  + Advantage: can be developed in a short amount of time.
  + Disadvantage: features in the prototype are more expensive to implement
* **Evolutionary prototyping**: use the prototype as basis for the implementation of the final system
  + Advantage: shorter time to market
  + Disadvantage: can be used only if the target system can be constructed in a prototype

*Risks of prototyping:*

* Developers may become attached to the prototype
  + Excessive development time of the prototype
* Customers might not appreciate how much work must be done to turn a prototype into a fully functional system
* Users may confuse prototype and finished system
  + The prototype may actually perform the functions it implements better than a fully functional system (e.g. due to reduced functionality, easier access)
* Negative feelings toward the software if the prototype has problems
* Cost of implementing the prototype

*Managing expectations:*

* “If you show a nonprogrammer a screen which has a user interface that is 90% worse, they will think that the program is 90% worse.”
* “If you show a nonprogrammer a screen which has a user interface which is 100% beautiful, they will think the program is almost done.”

*Storyboarding: modeling the user interface as finite automaton:*

* Makes it possible to navigate through the user interface design with the user
* Often the model can be use to generate code
* Allows to combine evolutionary prototyping and low-fidelity prototyping
* Example: storyboarding in Xcode

*Low vs. high fidelity prototypes:*

|  |  |  |
| --- | --- | --- |
|  | **Advantages** | **Disadvantages** |
| **Low Fidelity** | - Easy to produce  - More feedback  - No design decisions | - Mostly not reused  - Important details are ignored |
| **High Fidelity** | - More realistic  - More detailed problems can be identified  - More impressive | - May cause much effort  - Less feedback  - High expectations (especially with interactive Prototypes) |

*Summary:*

* Usability is important for the success of your software project
* Prototyping allows you to quickly evaluate the user interface design
  + The earlier you know that something does not work as intended, the faster you can improve
* Heuristics help to evaluate / test the usability of a system
  + Example: Nielsen’s 10 heuristics
* Benefits of prototyping
  + Prototypes allow to get early feedback —> save time while developing
  + Prototypes are easier to understand than text or diagrams —> easier communication
  + Prototyping improves the usability

**POM 10 – Software Configuration Management I**

*Why configuration management?*

* Multiple people work on artifacts that are changing
* More than one version of the artifact has to be supported
  + Released software systems
  + Custom configured systems (different functionality)
  + Systems under development
  + Software running on different machines & operating systems
* Need for coordination
* Software configuration management (SCM)
  + Manages evolving software systems
  + Controls the costs involved in making changes to a system

*What is software configuration management?*

Definition:

* **set of management disciplines** within a software engineering process to develop a **baseline**
* It encompasses (umfasst) the disciplines and techniques **of initiating, evaluating and controlling change** to work products during and after a software project

IEEE 828-2012: IEEE standard for configuration management in systems and software engineering

**Baseline**: a work product that can be changed only through a change control procedure

*Administering software configuration management:*

* Software configuration management is a **project function** with the goal to make technical and managerial activities more effective
* Software configuration management can be administered in several ways:
* Organization-wide
* Project-specific
* Distributed among the project members
* Mixture of all of the above

*Configuration management roles:*

* **Configuration manager**
  + Responsible for identifying configuration items
  + Often responsible for defining the workflows for creating promotions and releases
* **Change control board member**
  + Responsible for approving or rejecting change requests
* **Developer**
  + Creates promotions triggered by change requests or the normal activities of development
  + Checks in changes and resolves conflicts
* **Auditor**
  + Responsible for the selection and evaluation of promotions for release and for ensuring the consistency and completeness of this release

*Software configuration management activities:*

* **Configuration item identification**
  + Modeling the system as a set of evolving components
* **Promotion management**
  + The creation of versions for other developers
* **Build and Release management**
  + The creation of versions for clients and users
* **Change management**
  + The handling, approval & tracking of change requests
* **Branch management**
  + The management of concurrent development
* **Variant management**
  + The management of coexisting versions

*Terminology: configuration item:*

An aggregation of software, hardware, or both, designated for configuration management and treated as a single entity in the configuration management process

* **Software configuration items**: source files, models, tests, documents, configurations
* **Hardware configuration items**: CPUs, sensors, actuators

*Configuration item identification:*

Not every entity needs to be under configuration control all the time

Two Issues:

1. What: Selection of configuration items

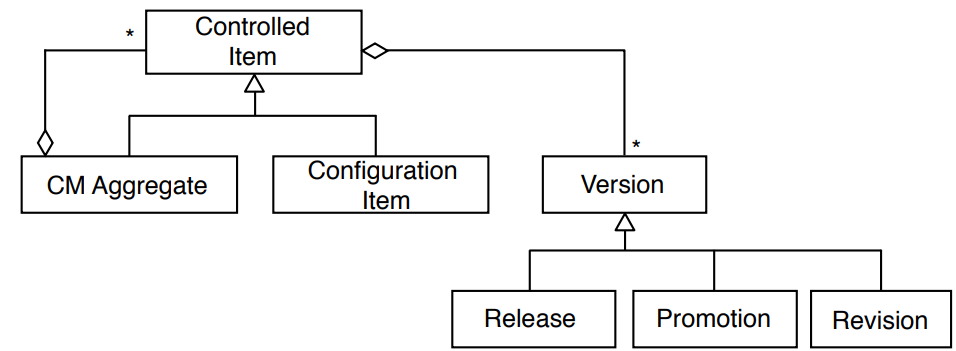
* What should be under configuration control?

1. When: When do you start to place entities under configuration control?

* In early days, it was an activity
* Nowadays it should be a project function

*Terminology: version:*

* **Version**: an initial release or re-release of a configuration item associated with a complete compilation or recompilation of the item
* **Release**: the formal (external) distribution of an approved version, e.g. a potentially shippable product increment in Scrum
* **Promotion**: a version that is made available (internally) to other developers, e.g. a new commit in the version control system
* **Revision**: change to a version that corrects only errors in the design/code, but does not affect the documented functionality

*Object model for configuration management (UML class diagram):*

*Terminology: baseline:*

* A specification or product that has been formally reviewed and agreed to by responsible management
* Thereafter it serves as the basis for further development, and can be changed **only** through change control procedures
* Examples
  + Baseline A: the API has been completely been defined; bodies of methods are empty
  + Baseline B: all data access methods are implemented and tested
  + Baseline C: the GUI is implemented

*Types of baselines:*

* As systems are developed, a series of baselines is developed, usually after a review (analysis review, design review, code review, system testing, ...)
  + **Developmental baseline**
  + **Functional baseline**
  + **Product baseline**
* Branch Management allows to transition between these baselines

*Naming Schemes for Baselines (Tagging):*

* Many naming scheme for baselines exist (1.0, 6.01a, ...)
* A 3 digit scheme is quite common:



*History of software configuration management tools:*

* RCS: The first on the block [Tichy 1975]
* CVS (concurrent version control)
  + Based on RCS, allowed concurrent working without locking
* Perforce
  + Repository server; allows to keep track of developer’s activities
* ClearCase
  + Multiple servers, process modeling, policy check mechanisms
* Subversion
* Git

*Version control systems (VCS):*

* VCS allow many software developers to collaborative work on the configuration items
* VCS store different versions of configuration items (e.g. source code and configuration data) in a commit history and allow to restore previous versions
* The commit history allows developers to see how the configuration items changed over time and to see who changed a certain item
* Revisions are stored in a repository and developers can check out a revision into a working copy
* Distributed version control systems (DCVS), also known as distributed revision control or decentralized version control, provide more flexibility and features

*Monolithic architecture for version control:*

* Developers have simple local database, keeps all changes to files under revision control
* Example: RCS (Revision Control System)
* Still distributed with many computers today

*Repository architecture for version control:*

* A single (central) server contains all the versioned files
* Developers check out files from the server to their computer, change them and check them back into the central server
* Administrators have fine-grained control over who can do what
* **Problem**: single point of failure in the central VCS server: possibility of loosing all the versions and their history if the server crashes

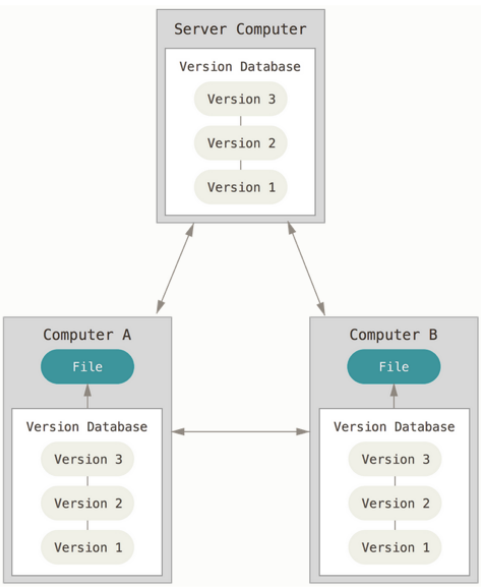
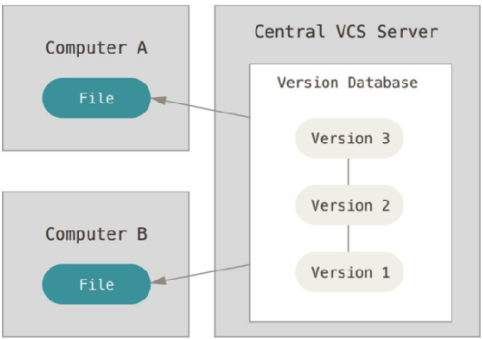
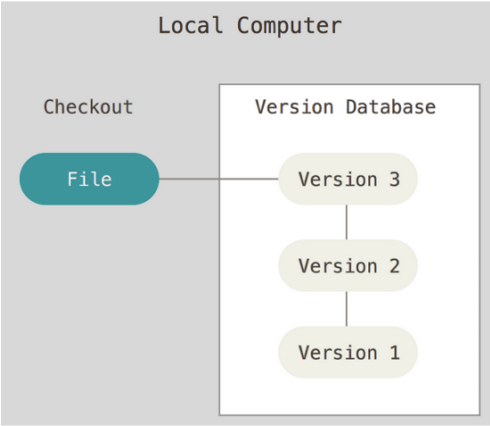
*Example of a repository architecture: Subversion:*

* Open source project
* Based on CVS (Concurrent Versions System)
* Distinction between programmer’s directory (working copy) and master directory (central repository)
* Commands:
  + Checkout: Check out a programmer’s working copy from the server repository
  + Add: Add a file to the programmer’s working copy
  + Delete: Delete a file in the programmer’s working copy
  + Commit: Commit changes from the programmer’s working copy to the server repository (create a new version and promote it)
  + Diff: Comparison between 2 versions
* The time for branch management is independent of the size of the system (unlike CVS, which creates physical copies of the files, Subversion uses only tags)

*Peer-to-peer architecture for version control:*

* Addresses the single point of failure problem
* Each programmer’s computer (Computer A, Computer B, …) fully mirrors the repository (Server Computer)
* Programmers can work offline and create versions (commits and branches)
* Not all versions are promoted to the master directory (they need to be pushed)
* If the server dies and a programmer has a full copy of the repository, it can be copied back to the server computer
* Example: git

**monotholic** **repository peer-to-peer**



*Example of a peer-to-peer architecture: git*

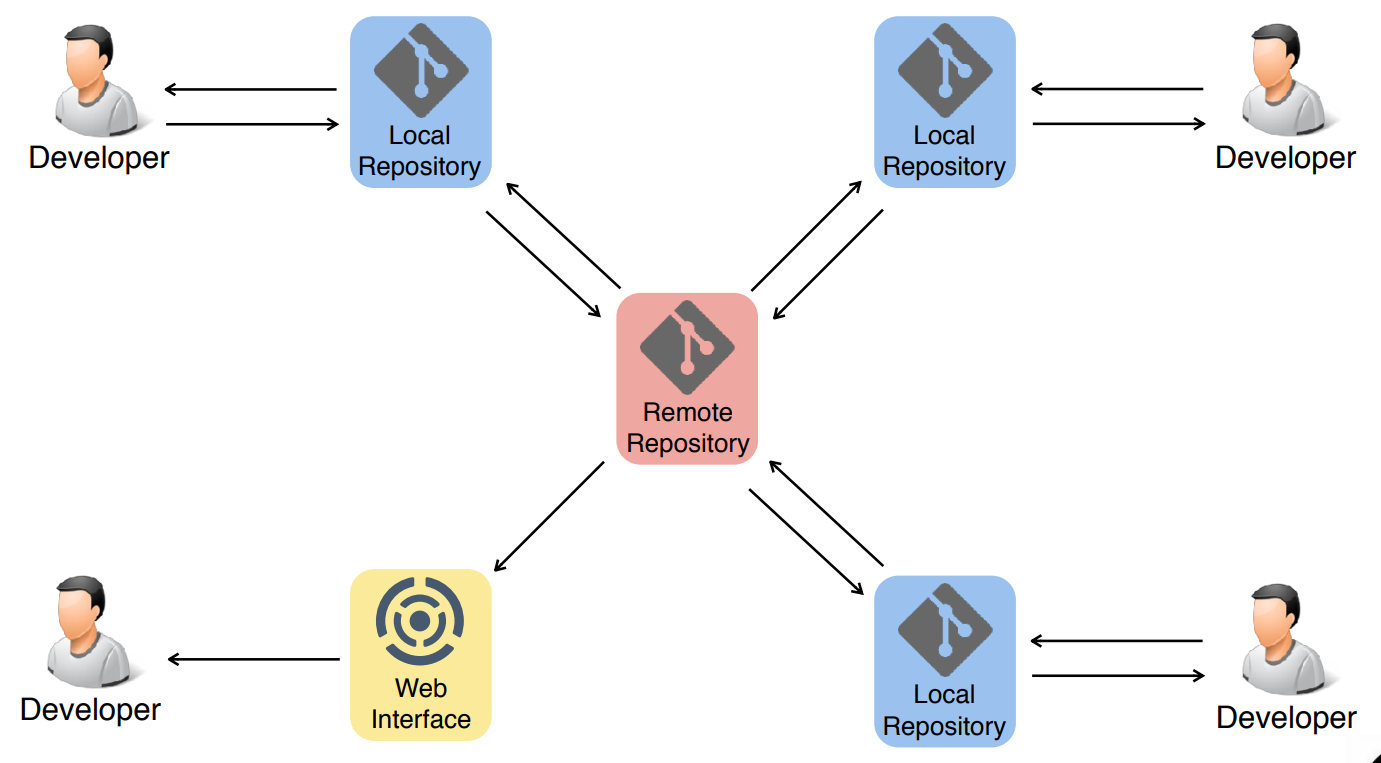
* Open source project
* Supports light-weight local branching
* Commands: clone, commit, push (=promote), fetch, merge, pull
* Differences to Subversion:
  + Support for multiple repositories, subversion supports only single repositories
  + Branches are light-weight



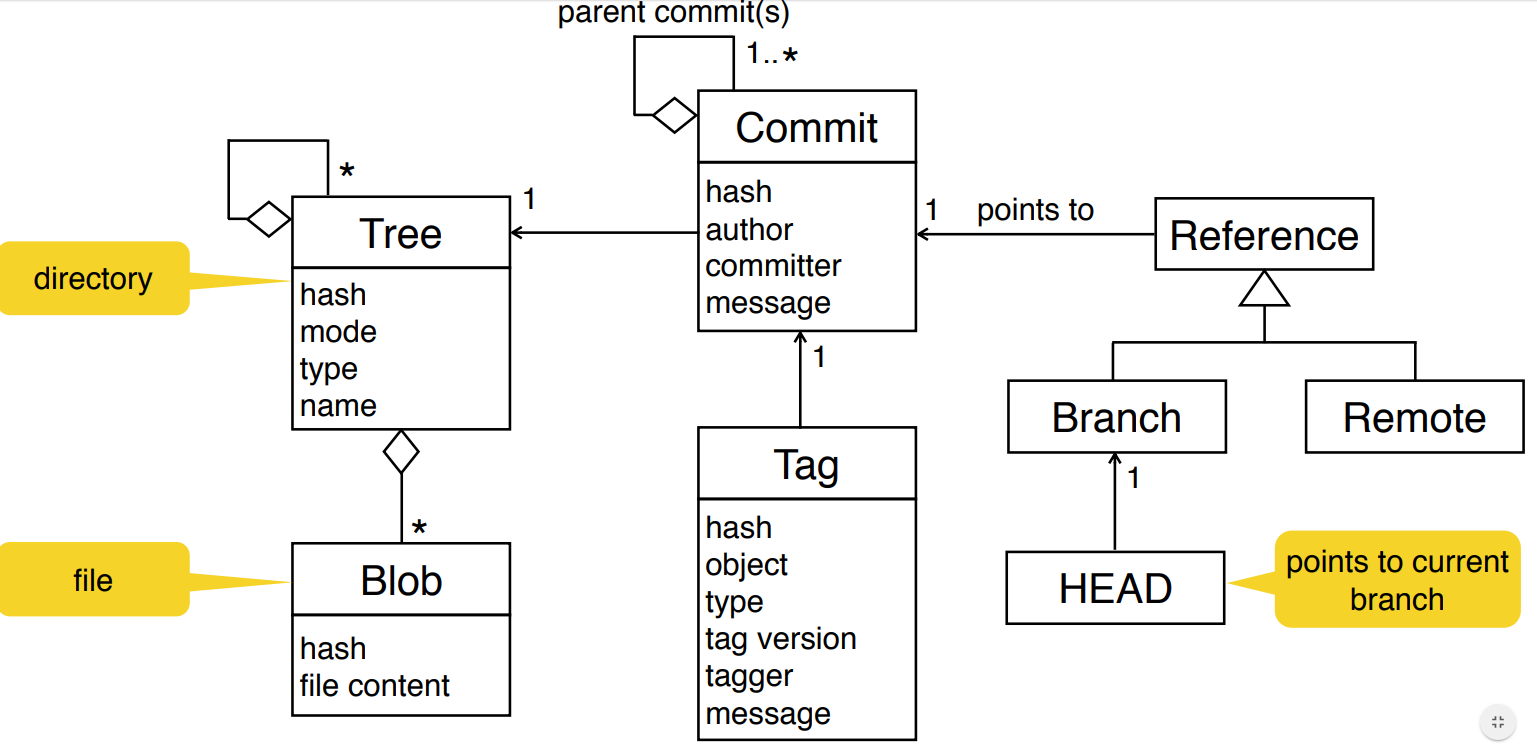
*Comparison of distributed vs centralized version control systems:*

* Advantages of distributed VCSs
  + Ability to work offline (local commits)
  + Ability to work incrementally (small commits)
  + Ability to context switch efficiently (lightweight branching)
  + Ability to do exploratory coding efficiently (lightweight branching)
* Disadvantages of distributed VCSs
  + High learning curve
  + Scaling issues
  + Less administrative control

**Distributed version control**



*Git object model (UML class diagram):*



*Resolve a merge conflict:*

1. Take mine (i.e. my changes and ignore the other changes)
2. Take theirs (i.e. the other changes and ignore my changes)
3. Merge the overlapping changes manually and decide per case possibly taking both changes

*Distributed version control:* ***best practices****:*

* Commit related changes (small commits, example: fixing 2 bugs should produce 2 commits)
* Commit and push often (share core more frequently with others)
* Do not commit half done work (commit code when it is completed)
* Test before you commit (… at least build and make sure it compiles)
* Write meaningful and understandable commit messages (summary, body)
* Do not use version control as a backup system
* Keep your working copy of the repository up to date (regularly pull and push)
* Use branches
* Agree on a workflow
* Do not change published (promoted) history

**POM 11**

**Change management**

• Change management is the handling of change requests

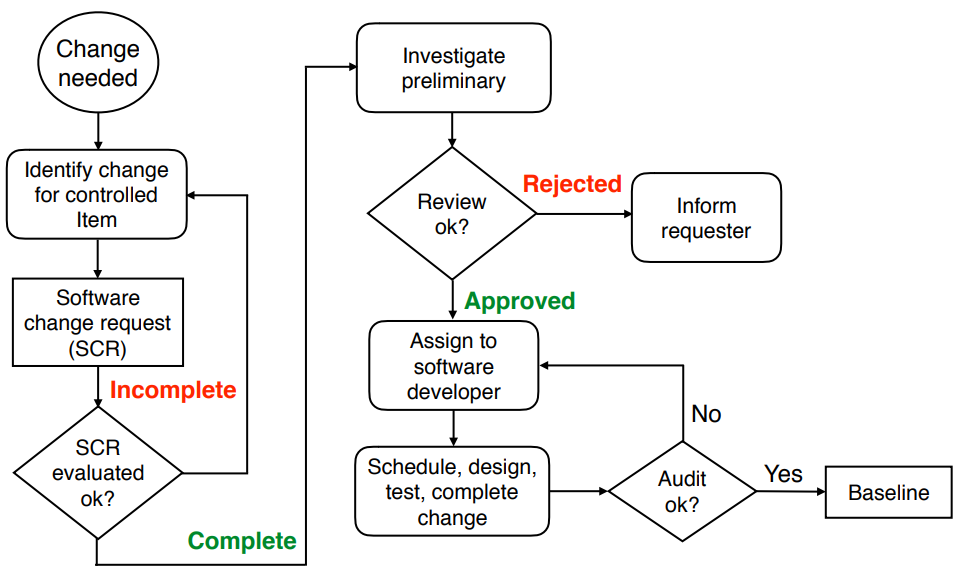
• The general change management process:

• The change is requested

• The change request is evaluated against requirements and project constraints and reviewed by the configuration control board

• Following these assessments, the change request is approved or rejected

• If it is approved, the change is assigned to a developer who will design, implement and test the change accordingly



**Change request**

Specifies the process for requesting a change to a configuration item and the documented informations:

* Nameand version of the configuration item where the need for change appears
* Originator’s (Gründer) name and address
* Date of request
* Indication of urgency (Dringlichkeit)
* Description of the requested change

**Change policies**

The purpose of a change policy is to guarantee that each promotion or release confirms to the accepted criteria.

Examples for change policies: “No developer is allowed to promote source code that was compiled with errors or warnings.”

**Change management activities and responsibilities**

1. Software configuration control: managing a change request

Define a change request form (formular)

Define management procedures for:

• Identification of the need for a change request

• Analysis and evaluation of a change request

• Approval (Genehmigung) or disapproval of a change request

• Implementation, verification and release of the change

1. Software configuration status accounting

Answers the following questions:

• What elements are tracked and reported for baselines and changes?

• What types of status accounting reports are generated? What is their frequency?

• How is information collected, stored and reported?

• How is access to the configuration management status data controlled?

1. Software configuration auditing (Überwachung)

• Includes the identification when, how and how often audits (Prüfungen) are necessary for the project

• An audit determines for each configuration item if it meets the required physical and functional characteristics

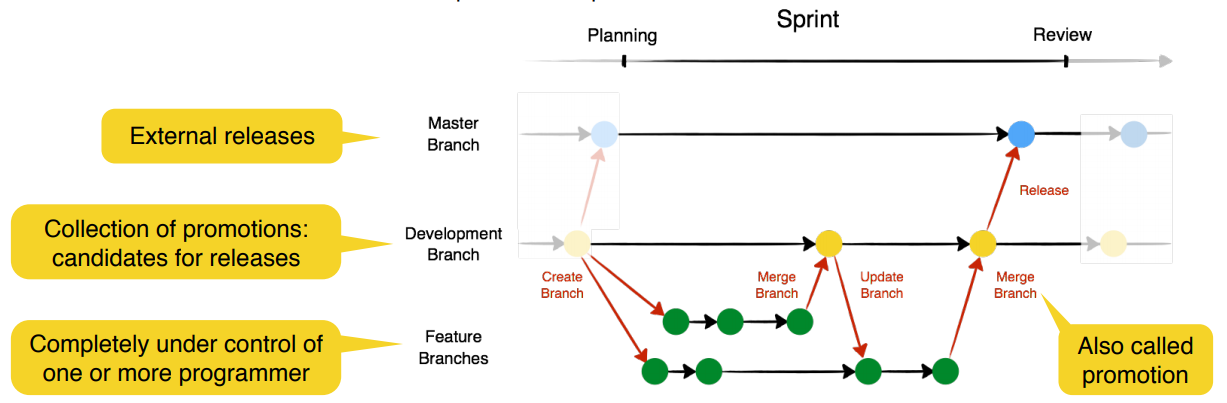
• Audits are conducted according to a well-defined process consisting of various auditor roles and responsibilities

➡Successful completion of an audit can be a prerequisite for the establishment of the product baseline

**Branch management**

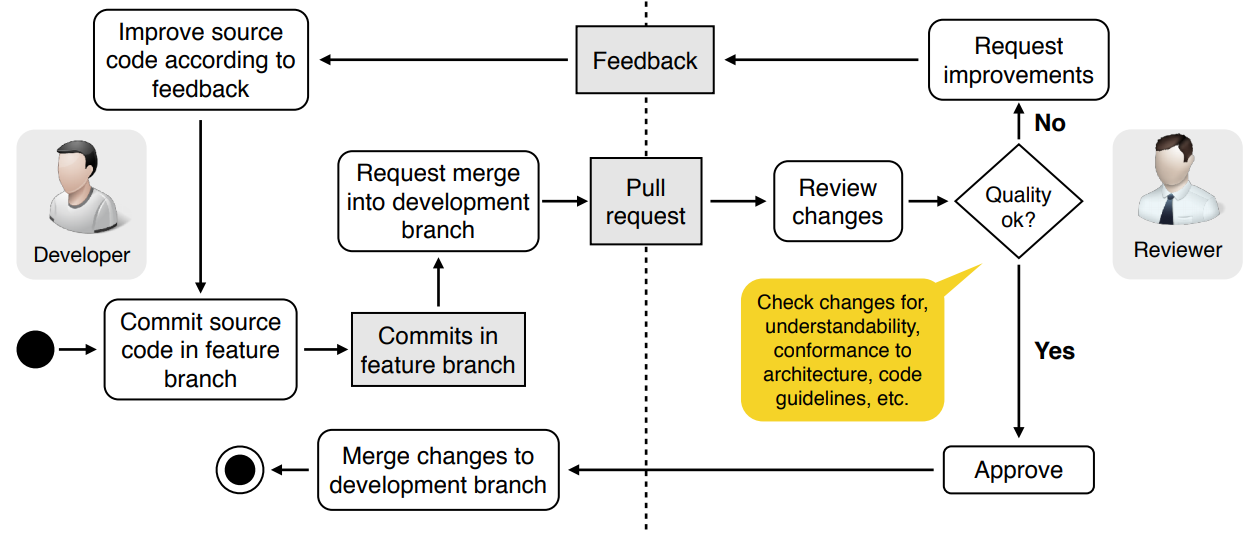
A branching model controls the concurrent development and defines rules when branches are created and merged (promoted)

* The **master branch** contains external release candidates (potential product increment)
* **Hotfix** **branches** include revisions (Verbesserung/Änderungen)
* **Release** **branches** contain external releases (to be used in production)
* The **development** **branch** contains internal release candidates
* **Feature** **branches** are used for the actual implementation work of new requirements but important: minimize feature branch lifetime (not more than “some” days)



**Merge management with pull requests and code reviews**

• Before the changes in a feature branch are merged (promoted) into the development branch, a reviewer looks at the changes and only approves the code if it is all right.



**The broken window theory**

Don't leave "broken windows" (bad designs, wrong decisions, or poor code) unrepaired.

One broken window, left unrepaired for any substantial length of time, instills in the developers of the code a sense of abandonment - a sense that the developers don’t care about the source code.

➡ Fix each broken window (i.e. bad source code) as soon as it is discovered

➡ Otherwise: clean, functional systems deteriorate (verschlechtern) quickly once windows start breaking

**POM 12 – Developing Winning Proposal**

*Definition Proposal:*

* legally binding
* formal and comprehensive offer to a client
* to solve a business problem
* by defining an appropriately tailored project

*General structure of proposals:*

* Understanding the client‘s problem
* Describing the solution to the problem
* Work plan (tasks, time, budget and personnel)
* Prerequisites (Voraussetzungen) for the client (client time, client personnel involvement,…)
* Pricing (possibilities 🡪
  + Fixed Price for fixed scope („all included“)
  + Bonus for ahead of time delivery
  + Value based billing)
* Legal issues 🡪 important part (Werkvertrag vs. Dienstvetrag, involve a lawyer,…)

1. Goal
2. Approach and Results
3. Provisioning of Infrastructure for the Project and participation Obligations
4. Project Organization
5. Milestones and Cost
6. Additional Agreements

*Winning proposal:*

* Understand your client`s intention and have him/her understand you
* Understand your client’s need and boundaries
* Build the **right** relationships (who is the real buyer, his/her motivation for the project)
* The team need a right skill-mix and has to be flexible to adjust
* Learn form every proposal – including failures
* Careful with the language of your proposal (best 🡪 good, highest 🡪 our standards, …)
* Do not forget: the proposal is only the first step of the project (promise only what you can deliver)
* offer a good price
* gain the trust of the client

*Other questions:*

**Value based billing:**

* The payment is proportional to the resulting financial benefits for the client

**Bidding for a proposal:**

* You currently do not have the required skills or capacity
* Similar proposals you made to this client were all rejected in the past

**Proposal process:**

* Analyze the client's needs & situation
* Build client relationships
* Test-run the proposal presentation

**POM Class 13**

**Requirements** **for build management**

Large and distributed software projects need to provide a development infrastructure with an integrated build management that supports:

* Regular builds from the master directory
* Automated execution of tests
* E-mail notification
* Determination of code metrics
* Automated publishing of the applications and test results

**Tools** for build management: Unix’s Make, Ant, Maven

The transition from **source code to the executable application** contains many mechanical activities. Executing these steps manually is time consuming and the chance of introducing failures is high -> Automating these steps

**Reasons for continuous integration**

* The later integration occurs in a project, the bigger is the risk that unexpected failures occur
* The higher the complexity of the software system, the more difficult is the integration of its components
* Continuous integration addresses these risks by integrating and testing as early and frequently as possible

**Begriffe**

* **Failure**: Any deviation of the observed behavior from the specified behavior (crash)
* **Error**: The system is in a state such that further processing by the system can lead to a failure
* **Fault**: The mechanical or algorithmic cause of an error (bug)
* **Validation**: Activity of checking for deviations between the observed behavior of a system and its specified behaviour

**How to deal with faults**

**Fault avoidance**

* Use methodology to reduce complexity
* Use software configuration management to prevent inconsistency
* Apply verification to prevent algorithmic faults
* Use reviews to identify faults already in the design

**Fault** **detection**

* Testing: Provoke failures in a planned way
* Debugging: Find and remove faults
* Monitoring: Deliver information about state and unusual behavior => Used during debugging

**Fault** **tolerance**

* Exception handling, modular redundancy

**Types of testing**

**Unit** **Testing**

Individual components (class or subsystem) are tested. Confirms that the component is correct

**Integration** **Testing**

Groups of subsystems and eventually the entire system are tested. Tests the interfaces of the subsystem

**System** **Testing**

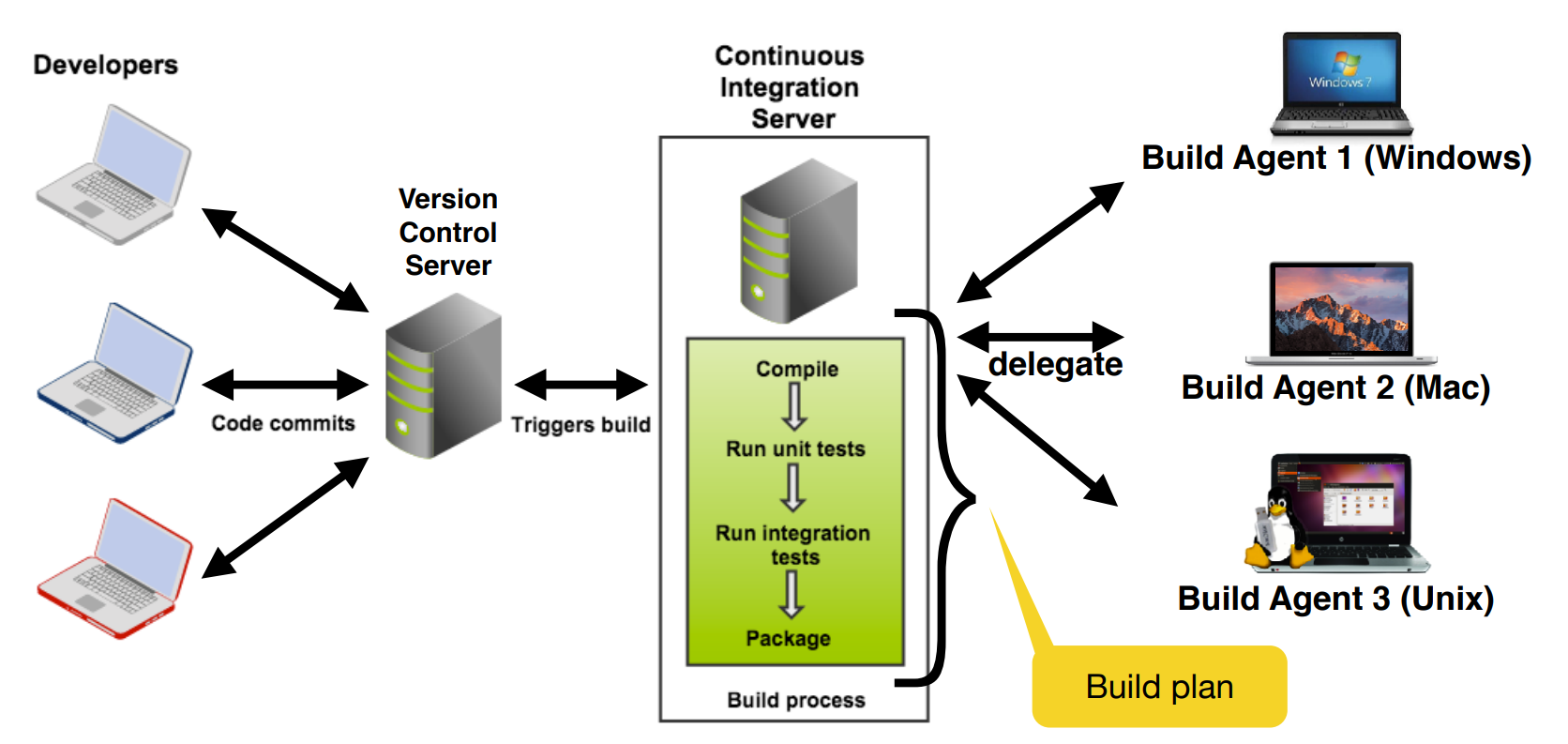
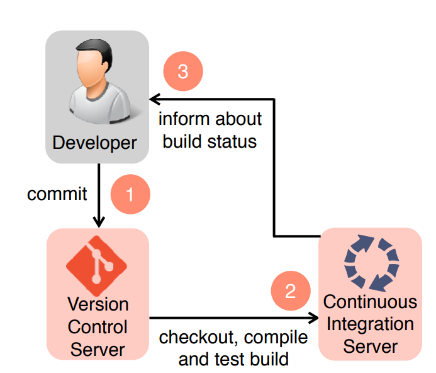
The entire system is tested. Checks if the system meets the requirements

**Acceptance** **Testing**

Evaluates the system delivered by developers (Carried out by the client). Checks if the system meets the requirements and if its ready to use

* **Alpha test** 
  + Client uses the software at the developer’s environment
  + Software used in a controlled setting, with the developer always ready to fix bugs
* **Beta test** 
  + Conducted at client’s environment
  + Software gets executed in a realistic target environment

**Continuous Integration:** A software development technique where members of a team integrate their work frequently. Usually each person integrates at least daily, leading to multiple integrations per day. Each integration is verified by an automated build which includes the execution of tests



**Advantages** of continuous integration

* There is always an executable version of the system
* Developers and managers have a good overview of the project status
* Automatic regression testing

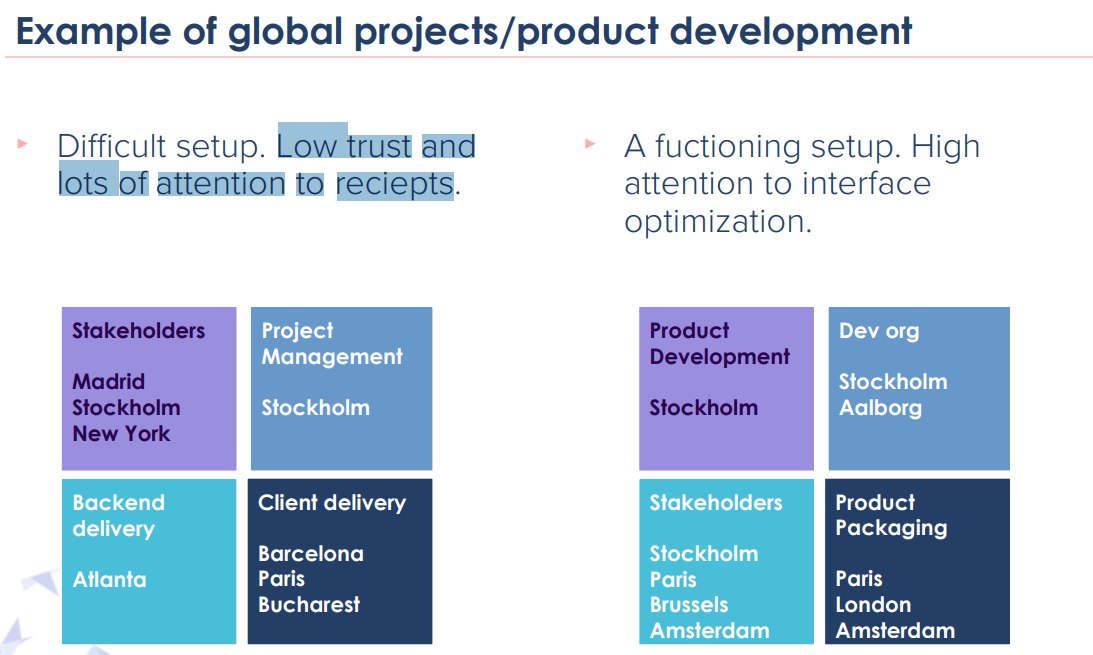
**Regression testing**

**Goal**: verify that software previously developed and tested still performs correctly even after it was changed or interfaced with other software

**Benefit**: finds errors in the existing source code immediately after a change is introduced

**Drawback**: can be very costly to execute a large test suite after each change

**POM 14**



**Communication**

* Its very important for a successful project delivery and product development
  + Impossible to deliver value without communication
  + Active & conscious (bewusste) communication will make most projects successful
* Lack of communication is one of the biggest reasons for people performing bad and not enjoying work

**Communication patterns**

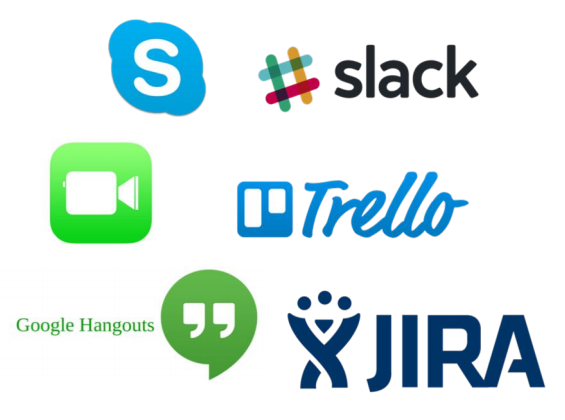
* Planned vs unplanned
* Synchronous vs asynchronous
* Vertical vs horizontal
* Formal vs informal
* Internal vs external stakeholders

**Communication challenges in global contexts and mitigation**

* Who to contact – roster
* Difficulty of initiating contact – document communication structure
* Language – explain back to each other
* Ineffective communication – be aware in communication strategy
* Lack of informal communication – build relationships

**Tools to enable communication ???**

* Invest in tech
* Plan for online & offline
* Technical issues
* Office facilities
* Combining live communication and remote
* Tracking of communication



**Travel!**

* Invest in meeting face to face
* Have a satellite in the other location, channeling things
* When budgeting the project always argue for the travel budget
* Compensate for climate effects!

**Practical tips on communication in global contexts**

* Focus on more planned communication
* Make sure to schedule time for synchronous communication
* Easily accessible meeting notes
* Invest in tools/facilities to enable informal communication
* Create a mindset of awareness and respect of time difference
* Invest in travel

**Differences in product development vs projects**

|  |  |
| --- | --- |
| Product development | Projects |
| * Done with long term organization and with no decided end date   + Invest in long term productivity | * possible to create small teams for a short period of time for short term delivery   + high investment – high energy |
| * adds dependencies by size and product lifecycle   + IT architecture is important | * can more easily free themselves from patterns and habits and focus on the project goal |

**Architecture and infrastructure for global contexts**

* Think smart when “Slicing the Elephant”
  + Split the responsibility in small parts to keep the dependencies low
* If possible, choose an architecture that enables distributed teams
* Continuous integration to avoid planning dependencies



**Practical tips to enable agile methodologies in a global setup**

* Add extra collaboration forums
* Keep teams in one location
* Make sure the teams know about each other and share knowledge between teams
* Make sure to plan for daily cooperation business people

**Collaboration Forums ????**

* Team meetings (daily standups, retros, reviews, etc.)
* Alignment meetings (scrum of scrums, po forums, stakeholder forums, etc.)
* Status meetings (demos, planning sessions, etc.)
* Knowledge forums (presentations, workshops, brainstorming sessions, etc.)
* Networking forums (team breakfasts, lucky lunches, hangout coffees, etc.)

**POM 15 – Estimation and Scheduling**

**Estimation and Scheduling (Schätzung und Planung)**

*Facts about Estimates:*

* Nearly 2/3 of projects significantly overrun their cost estimates
* The average project exceeds its schedule by 100%

*Challenges for Estimation:*

* **Incomplete knowledge** about:
  + Project scope and changes
  + Prospective resources and staffing
  + Technical and organizational environment
  + Infrastructure
  + Feasibility of functional requirements
* Comparability of projects in case of new or changing technologies, staff, methodologies
* Learning curve problem

*Components of estimations:*

* **Cost** (Personnel, Material, Extra costs (travel expenses etc.))
* **Scope** (Number of requirements, Complexity of requirements)
* **Time** (Development Time, Project duration, Dependencies)
* **Quality** (Usability, Maintainability, Reliability, ...)

and also…

* **Infrastructure** (rooms, technical infrastructure)



*Estimating Personnel Cost:*

* Personnel type: Team leader, analyst, designer, programmer, tester…
* **Cost rate**: Cost per person per day
* 2 alternatives for cost rate:
  1. Single cost rate for all types
  2. different cost rates to different personnel types based on experience, skills, …
* **Personnel cost**: person days x cost rate

*Estimating Development Time:*

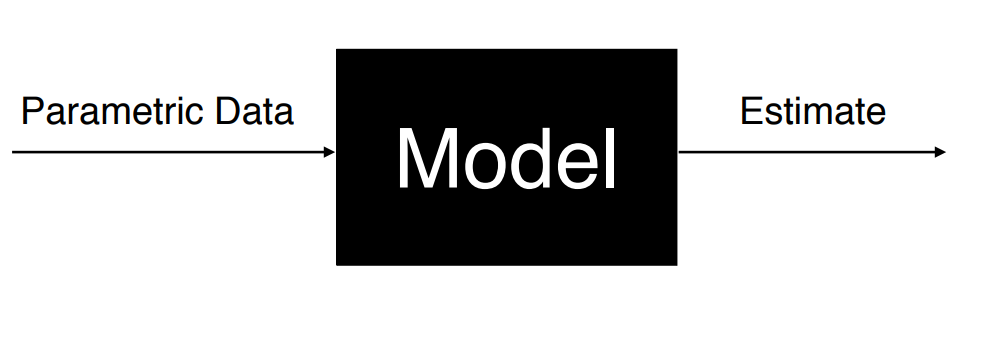
* Development time is often estimated by the formula

**Duration = Effort / People**

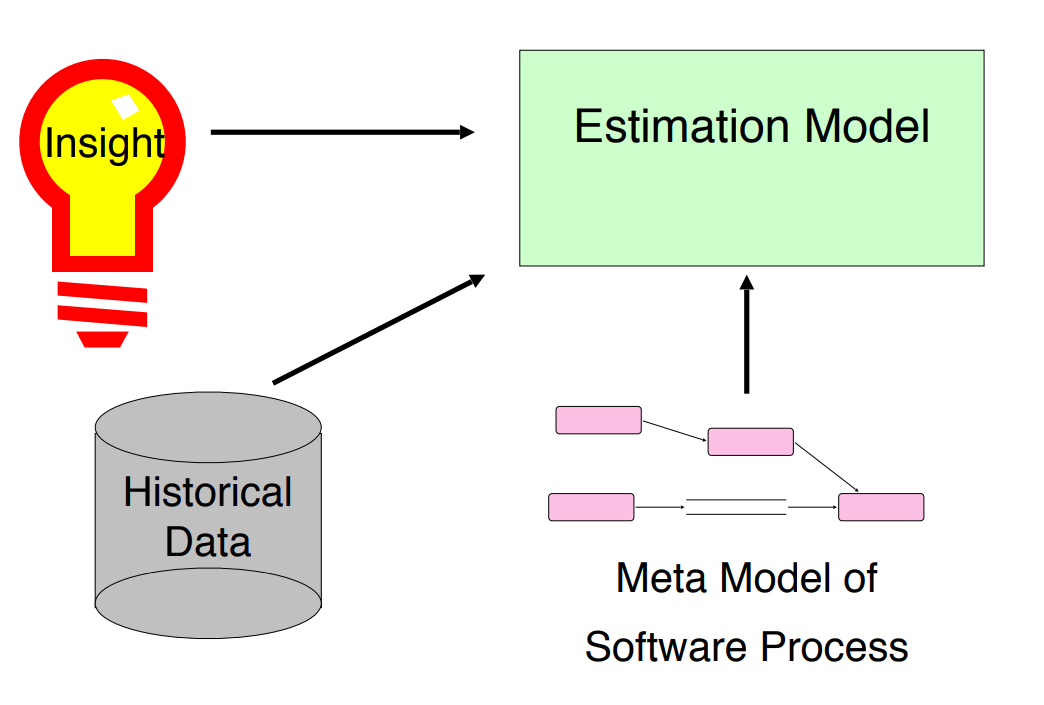
*Problems with this formula:*

* larger project team increases communication complexity which usually reduces productivity
* not possible to reduce duration arbitrarily (willkürlich) by adding more people to a project
* **Brooks Law:** “Adding people to a late project makes it even later”

*Basic Estimation Model:*



*How do you Build an Estimation Model?*



*Top-Down and Bottom-Up Estimation:*

2 common approaches for estimations

1. Top-Down Approach
   * Estimate effort for the whole project
   * Breakdown to different project phases and work products
   * If you did not break down the work before
2. Bottom-Up Approach
   * Start with effort estimates for tasks on the lowest possible level
   * Aggregate the estimates until top activities are reached
   * Preferred if you already have activities and tasks and knowledgable developers

*Methods for estimating cost and effort:*

|  |  |
| --- | --- |
| Expert estimation | Algorithmic estimation |
| Based on:   * Experience * Domain knowledge | Based on:   * Key performance indicators * Formulas * Metrics |
| Exemplary methods:   * T-Shirt sizes * Planning Poker * Assessment meetings | Exemplary methods:   * Lines of Code (LOC) * COCOMO II * Function Points |

*Expert Estimation:*

* Guess from experienced people
* No better than the participants
* Suitable for atypical projects (für atypische Projekte geeignet)
* Justification of the result difficult (Begründung des Ergebnisses schwierig)
* Important when no detailed estimation can be done
* Works best if the estimates are for short term items

*Algorithmic Estimation:*

* Calculations using input variables
* needs information about key perfomance indicators
* easier to justify results
* Suitable for traditional projects

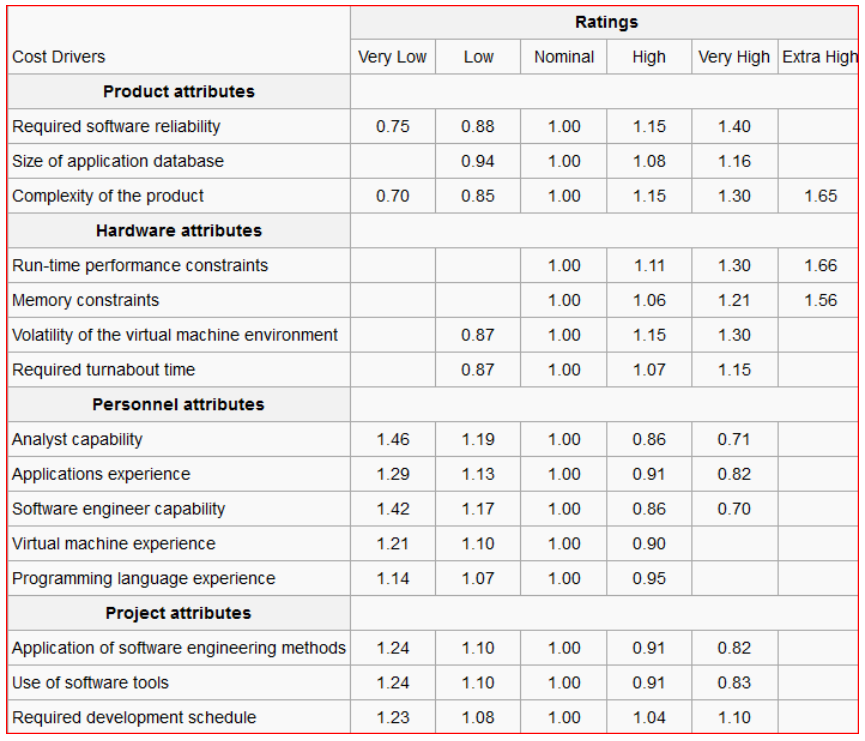
*Lines of Code:*

* Traditional way for estimating application size
* Advantage: easy to do
* Disadvantages:
  + Focus on developer’s point of view
  + No standard definition for “Lines of Code”
  + **programmers ignore reuse and refactorings**
* Caspers Jones: The use of LOC metrics for productivity should be regarded as professional **malpractice**

*COCOMO (COnstructive COst MOdel):*

* Developed by Barry Boehm in 1981
* Top-down approach to estimate cost, effort and schedule of software projects, based on size and complexity of projects
* Assumptions:
  + Derivability of effort by comparing finished projects (“COCOMO database”)
  + System requirements do not change during development
  + Exclusion of some efforts (for example administration, training, rollout, integration)
* COCOMO II: Revision of COCOMO I in 1997
  + Targeted for iterative software lifecycle models such Boehm’s spiral model (COCOMO I was based on the waterfall model)
  + It also addresses team experience, developer skills and distributed development

*Cost Drivers in COCOMO II:*



*Estimation Variability: Cone of Uncertainty:*

*Problems with Traditional Estimation Techniques:*

* Focus on the completion of activities and not on the delivery of features
  + Customers get no value from the completion of activities
* Wrong focus in schedule reviews
  + The reviewers look for overlooked activities, not for overlooked features
* When faced with overrunning a schedule, teams
  + Attempt to save time by reducing quality

*Review: T-Shirt sizes:*

* Estimation is particularly challenging for unexperienced developers and in the beginning of a project
  + Example: The 1st sprint planning usually needs a lot of time due to estimation
* Recommendation: start with a simplified model, e.g. with T-Shirt sizes
* You can map T-Shirt sizes to story points, e.g.
  + S —> 1
  + M —> 3
  + L —> 8
  + XL —> too big, split it into smaller issue
* When the developers get more familiar with estimation and the project itself, move to a more fine-grained estimation model, e.g. Planning Poker

*Planning Poker: an agile estimation approach:*

* Planning poker packages expert estimates into an enjoyable approach to estimating
* All the participants are estimators, in particular all the developers, not just the project manager!
* The estimation team should not have more than 10 people
  + If there are more than 10 people, create two or more estimation teams
  + The product owner can participate in the game, but does not estimate
* Most important aspect of planning poker:
  + Estimates are arrived at by **consensus building**, not by looking at a crystal ball

*Planning Poker Rules:*

1. Start: Each team member is given a set of cards
2. One person reads each of the items to be estimated

* Examples of items: Functional requirements, user stories, scenarios, features, design goals, implementation of algorithms, TODOs

1. The team discusses each item individually
2. Each team member estimates the difficulty of the item and privately selects a card representing the estimate
3. After all team members have chosen their card, everyone shows their chosen card to the others at the same time

* If the team member’s estimates match, the estimate for this item is established (for now)

1. If estimates are not the same, the group discusses the differences

* Steps 4 to 6 are repeated until a consensus is reached
* Important: Do NOT average during planning poker

*Challenges and advantages of planning poker:*

* **Challenges**:
  + based on expert opinion
  + same backlog provided to another Scrum Team, it could come up with estimation that vary drastically different
  + Estimation changes with the different skills and experience
  + Team estimates while individuals are owners
  + “What” factor when not well described makes it harder for the team to estimate.
* **Advantages**:
  + Group Estimating: wisdom of the Crowd
  + The conversation following the revealing of initial estimates is a is a great way to pool important issues
  + Story points can be used to determine the Sprint velocity

*Planning Poker:* ***Practical difficulties:***

* Many people in a team may lead to long discussion
* Lazy developers want to overestimate tasks, in order to have less user stories assigned to them in a sprint
* People tend to want to average the amount of points, which limits the discussion and may not represent the actual difficulty of the task

***Tips*** *for Planning Poker:*

* Keep discussions productive:
* Break out into smaller sessions
* Choose the right time to play
* Write down the reason for the awarded points
* Consider the level of uncertainty
* Distribution of workload according to skill

*Dependency Diagrams:*

**A dependency diagram is a formal notation that helps in the construction and analysis of complex schedules.**

Example:

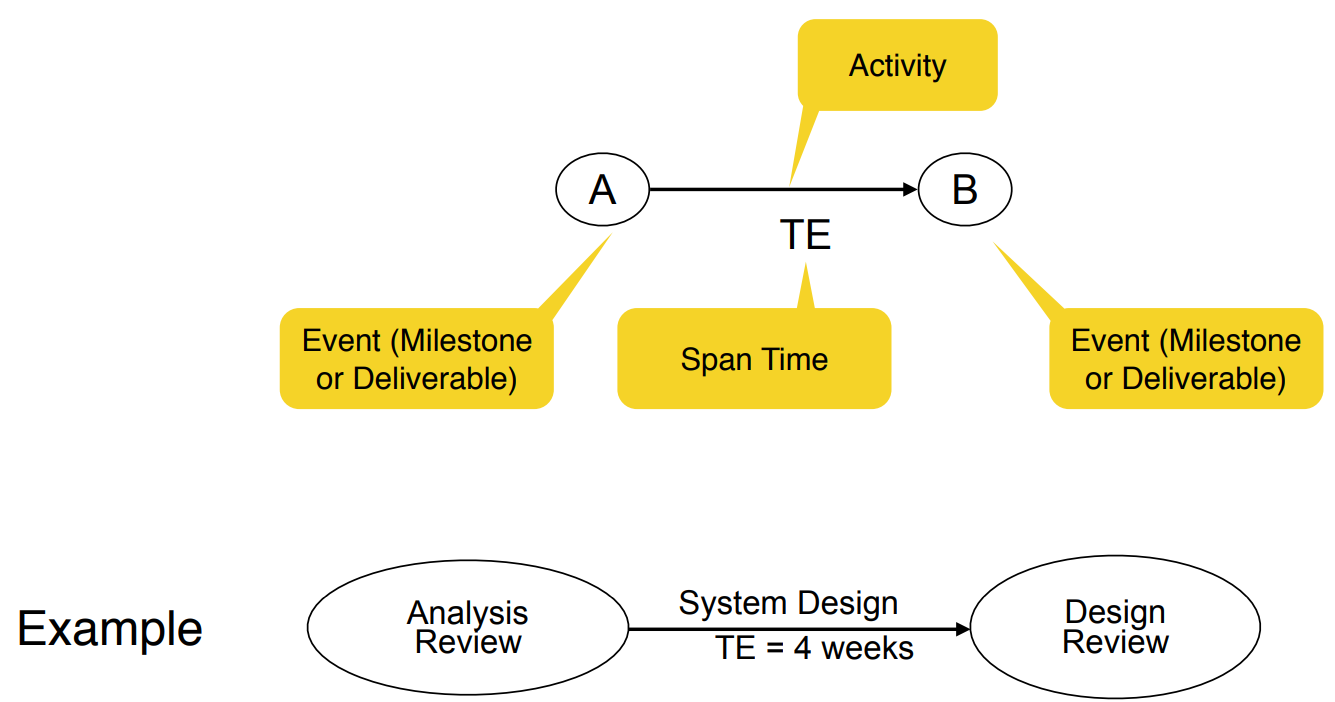
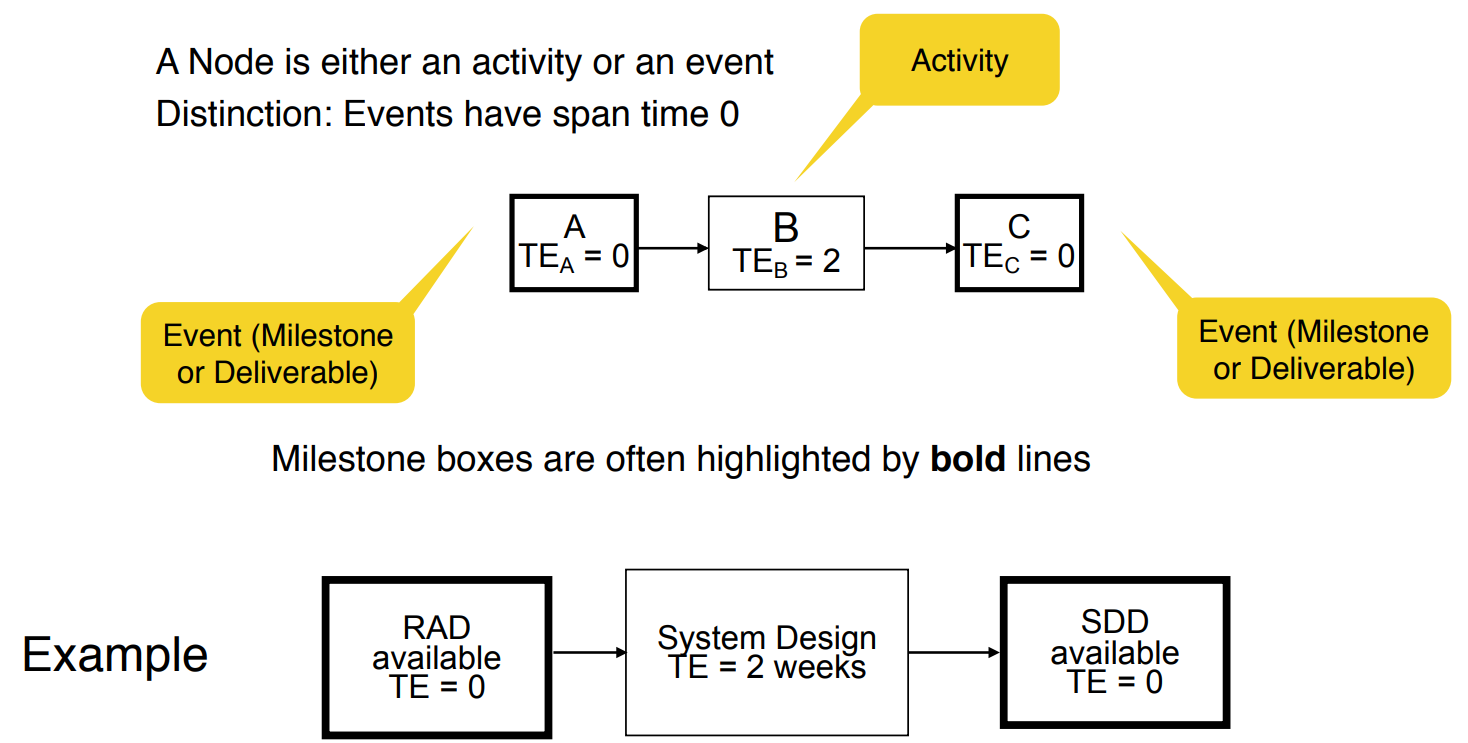
* You are assigned a project consisting of 5 tasks
* Task 1 has to be finished before any other task can start
* Task 2 and 3 can be done in parallel, task 4 and 5 cannot
* Task 5 depends on task 2
* Can the project be finished in 3 weeks, if each of the tasks takes a week to complete?

*Dependency Diagrams Overview:*

* Dependency diagrams consist of 3 elements
  + **Event**: A significant occurrence in the life of a project
  + **Activity**: Amount of work required to move from one event to the next
  + **Span time**: The actual calendar time required to complete an activity
    - parameters: availability of resources, parallelizability of the activity
* Dependency diagrams are drawn as connected graphs of nodes and arrows. 2 commonly used notations are:

1. Activity-on-the-arrow notation
2. Activity-in-the-node notation

*Activity-on-the-arrow Diagram Notation: Activity-in-the-Node Diagram Notation:*

*What do we do with these diagrams?*

1. Compute the project duration
2. Determine activities that are critical to ensure a timely delivery
3. Analyze the diagrams
   * To find ways to shorten the project duration
   * To find ways to do activities in parallel

* 2 techniques are used

1. Forward pass analysis (determine critical path)
2. Backward pass analysis (determine slack time)

*Critical Path and Slack Time:*

* **Critical path**
  + A sequence of activities that take the longest time to complete
  + The length of the critical path defines how long a project will take to complete
* **Noncritical path**
  + A sequence of activities that can be delayed and the project can still finish in the shortest time possible
* **Slack time**
  + The maximum amount of time that you can delay an activity and still finish your project in the shortest time possible

*Analyzing Dependency Graphs:*

* Determination of **critical paths**
  + Compute earliest start and finish dates for each activity
  + Start at the beginning of the project and determine how fast you can complete the activities along each path until you reach the final project milestone
  + Also called **forward pass analysis**
* Determination of **slack times**
  + Start at the end of your project, figure out for each activity how late it can be started so that you still finish the project at the earliest possible date
  + Also called **backward pass analysis**

*Definitions: Start and Finish Dates:*

* **Earliest start date** (ES): The earliest date you can start an activity
* **Earliest finish date** (EF): The earliest date you can finish an activity
* **Latest start date** (LS): The latest date you can start an activity and still finish the project in the shortest time
* **Latest finish date** (LF): The latest date you can finish an activity and still finish the project in the shortest time