Table of Contents

Introduction 1.1

1. Lectures

01 Project Organisation 2.1
02 Software Lifecycle Models I 2.2
03 Scrum 2.3
04 Software Lifecycle Models II 2.4
05 Usability Management I 2.5
06 Usability Management II 2.6
07 Software Configuration Management I 2.7
08 Developing Winning Proposals 2.8
09 Kanban 2.9
10 Software Configuration Management II 2.10
11 Contracting 2.11
12 Estimation and Scheduling 2.12
13 Build Management 2.13
14 Global Project Management 2.14
15 Agile Architecture 2.15
16 Release Management 2.16
17 Risk and Demo Management 2.17
18 Project Management Antipatterns 2.18

2. Exercises

3. Questions

Quiz Questions 4.1

POM Lecture Summary

Course Summary, Summer Term 2017

> **tl;dr:** This purpose of this document to collaborativeley create a both concise and detailed course summary of the *POM* Lecture from 2017 Summer Term at TUM.
>
> To learn as effective as possible, I would like to encourage everyone to engage in the discussion evolving around the content of this document. If you have questions or challenge what someone else wrote please do so in a **constructive way**. We are all new to the subject of Protein Prediction and mistakes happen. Let's learn from them together!

Official Lecture Resources

Lecture Homepage: Web

Lecture Resources: Moodle

Getting Started

This document is set up a **Gitbook** and hosted on **Github**. When you read this, you were already granted access to the repository so the first step is done.

The easist way to start contributing is to download **Gitbook Editor** (available for Mac, Linux, Windows) from here.

Before you add / change anything, please read through the Contribution Guide.

Contribution Guide

> Tell others what you work on | Write meaningful commit messages | Push often | Use American English

Why is there a contribution guide? I think it is in everyone's best interest to keep this summary as easy to understand as possible for everyone. This guideline should help to maintain consistency across the entire document.

Each section may contain a short additional information on how to format things specific to that section. Please have a look there as well.

1. Adding new content

1.1 Adding minor updates

If you add minor updates, like the answer to a single question, you can do this on the `develop` branch directly. Make sure your commit has a meaningful message.

1.2 Adding major updates

If you add major updates, like several related changes (e.g. an entire lecture summary), go along as follows:

1. Add a new **issue** on Github, describing what you are working on
2. Create a `feature/<issue-name>` branch and add your changes
3. Open a pull-request to merge back into `develop` and add the other contributers as reviewers
4. Once the pull request is merged, delete your feature branch and close the issue by referencing the merge commit

Why so complicated? This way the issues reflect new changes and are transparent for all contributors.

2. Challenging existing content

If you find obvious mistakes (typos, clearly wrong statements) just change them directly.

If you are challenging statements, answers to questions etc. which might not be trivial to understand go along as follows:

1. Open a new **issue** on github.
2. Reference the the statement in question you consider to be wrong
3. Provide an explanation why you think it is wrong
4. Provide your correct solution.

3. Adding new contributors

The purpose of this document is to foster collaborative learning - hence to make this as inclusive as possible. This being said, too many collaborators would probably lead to chaos . If you know other students personally, you want to add to the project shoot me a message and we will figure it out.

01 Project Organisation

3 May - 9 May

Slides: Introduction

Slides: Team Building

Slides: Project Organization

Project:

- is an undertaking, limited in time, with a clear goal and a specific budget, requiring a concerted effort
- consists of:
  - start date and duration
  - deliverables
  - schedule
  - activities
  - resources

Role

- defines a set of responsibilites

Project manager

- Administer resources
- Make sure the project goals are met

Analyst

- Analyse application domain
- Create a taxonomy of the domain abstractions

System architect

- Decompose the system into subsystems
- Choose a software architectural style

Meeting Management Roles

- **Primary facilitator**: organize the meeting, guide the execution, set up agenda

- **Minute taker**: record the meeting, identify action items and issues, distributes the minutes
- **Time keeper**: make sure time limits are kept

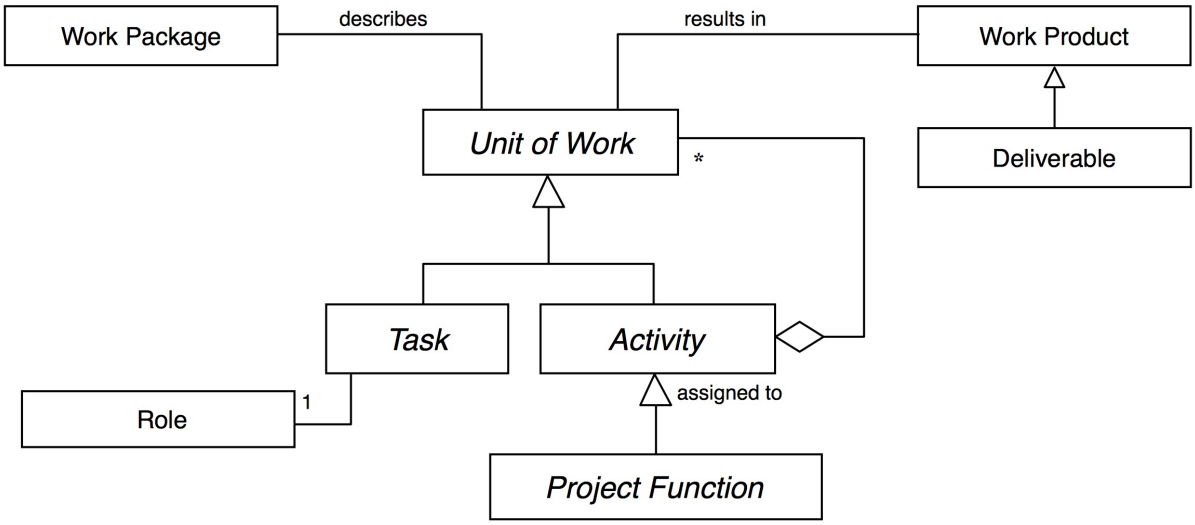
Key Concepts for Mapping Roles to People

- authority
- responsibility
- accountability
- delegation
  - reasons for delegation:
    - time management
    - expertise
    - training

Task - Activity - Unit of work

Task	Activity	Unit of work
smallest amount of work monitored by the project manager	consists of many tasks; activities are usually project milestones	higher level activity (bigger activity that can consist many smaller activities)

Project Organization Structures



Project Function: an activity that spans the entire duration of a software project

Work Package:

- describes a task/activity
- the work product to be produced by that task/activity
- acceptance criteria for it

Work Product: the visible outcome of a unit of work

Project Organization Structures

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

Organization forms

Functional Organization

- In a functional organization people are **grouped into departments** (Finance, production, sales / Analysis, design, testing)

Advantages	Disadvantages
Department members have a good understanding of what they are doing	high chance of overlap/duplicated work

Project-based Organization

- people are assigned to a project, each of which has a problem to be solved
- When to use it:
  - requirements expected to change during the project
  - project has a high degree of uncertainty

Advantages	Disadvantages
New people can be hired who are familiar with the problem or have special capabilities	teams cannout be assembled rapidly
responsive to new requirements	Roles/responsibilities have to be redefined for each project

Matrix Organization

- People from different departments of a functional organization are assigned to work on one or more projects
- Challenges:
  - double boss problem (department manager vs. project manager)
  - department vs project interests
  - multiple work procedures/reporting systems

Advantages	Disadvantages

Teams for projects can be assembled rapidly from departments	Team members are not familiar with each other
expertise can be applied to different projects as needed	Team members have different working styles

Gradual-Staffing: hiring people as needed

Flat-Staffing: all participants are assigned at the start of the project

Why Non-hierarchical project structure?

- Cutting down on bureaucracy
- Reduces development time
- Decisions are expected to be made at each level Negative aspect: hard to manage

Communication

- In large system developments you will spend more time communicating than coding

Soft skills:

- **Collaboration**(negotiate requirements with the client and with team members)
- **Presentation**(present parts of the system during reviews)
- **Management**(manage the team meetings)
- **Technical writing**(write project documentation)

Communication Event vs. Communication Mechanism

Communication Event	***Communication Mechanism***
= information exchange with defined objectives and scope - schedules events: review, meeting - unscheduled events: request for change, clarification, bug report	= tool that can be used to deal with communication events - synchronous mechanism: requires partners to be available at the same time (formal, informal meeting) - asynchronous mechanism: does not require partners to communicate at the same time (e-mail, newspaper, wikis)

Communication Anti-pattern:

- E-mail
- refactored solution: Slack

Software Project Management Plan (SPMP)

- The controlling planning document for a software project
- Specifies the technical and managerial approaches to develop the software product
- Companion document to the requirements analysis and system design documents
- May be part of the project agreement
- Very useful at the beginning of the project

Project Agreement (contract, statement of work, business plan)

= a document written for a client that defines the scope, duration, cost, deliverables, quantities, delivery
dates, delivery location

- Client: individual or organization that specifies the requirements and accepts the project deliverables
- Deliverables: work product to be delivered to the client – documents, demonstration of requirement

02 Software Lifecycle Models I

10 May - 16 May

Slides: Moodle

1. Software development as Application Domain

Methodologies: provide general principles and strategies for selecting **methods** and **tools** in a given **project environment** when errors occur.

1.1 Introduction

Inherent Problems with Software Development

- Requirements are constantly changing / not known in advance
- Frequent changes are difficult to manage (checkpoints for estimation & planning are difficult to identify)
- Several Systems of different maturity need to be considered

Why do we need Software Lifecycle Models?

- Development of large systems requires *organized approach*
- Software Management can be seen as complex system
- Methods to model systems (e.g. UML) can be used to model Software Management

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.

1.2 Modeling

- **Advantages**
  - enable developers to deal with complexity
  - make implicit knowledge about the system explicit
  - formalize knowledge and make it 'shareable'
- **Disadvantages**
  - models can themselves become very complex

Where do we need Models?

- **[A] Communication:** Models provide a common vocabulary. Informal models can be used to communicate an idea.
- **[B] Analysis / Design:** Models enable developers to reason about a future system.
- **[C] Archival:** Compact representation of the design and rational of a system (documentation).

[A] Models that support Communication

- called *Communication Models*
- purpose: communicate and idea to another **person** (support communication!)
- used in early phases of the project, during informal meetings
- does NOT need to be consistent, complete or have correct notation
- e.g. napkin model, whiteboard, mockup

[B] Models that support Analysis / Design

- called Specification *Models*
- purpose: provide a representation to reason about the system
- used to communicate an idea to a **tool**
- needs to be consistent and complete
- needs to have correct notation, so it can be entered into CASE tool
- UML is the preferred notation for specification models

1.3 Managerial Challenges of Modeling

Formalizing Knowledge is expensive

- takes time and efforts from developers
- **Antipattern:** Analysis Paralysis
- requires validation and consensus

Models introduce redundancy

- If the system is changed, the model needs to be changed
- If several models (different aspects) exist, several models need to be changed
- If a model is out of sync, it looses it's value

Models become complex

- With growing complexity, models use their value. If the model is similarly complex as the system, why use it?

Typical Software Lifecycle Questions

- Which **activities** should we select for the software project?
- What are the **dependencies between activities**?
- How should we **schedule the activities**?

Software Development Activities (Examples)

- **Requirement Anlysis:** What is the problem?
- **System Design:** What is the solution?
- **Detailed Design:** What are the best mechanisms to implement the solution?
- **Program Implementation:** How is the solution constructed?
- **Testing:** Is the problem solved?
- **Delivery:** Can the customer use the solution?
- **Maintenance:** Are enhancements needed?

2. Modeling the software lifecycle

2.1 Tailoring

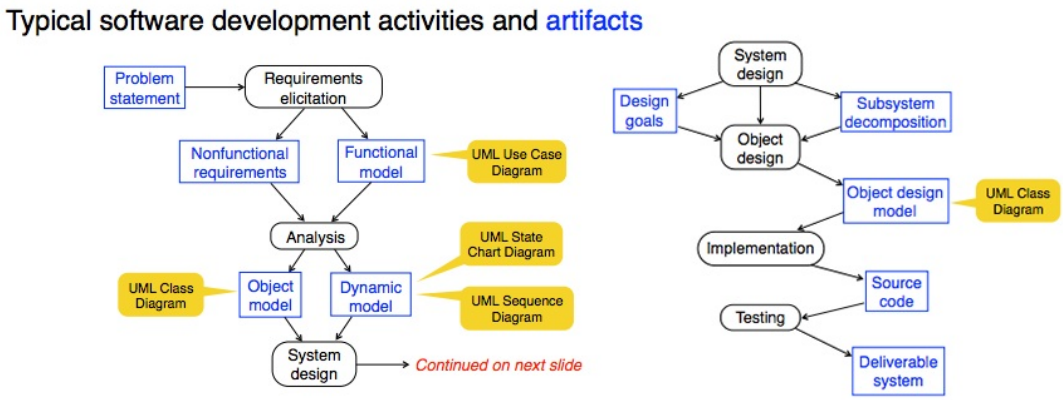
There is no “one size fits all” software lifecycle model that works for all possible software engineering projects

Tailoring: Adjusting a lifecycle model to fit a project

- **Naming**: Adjusting the naming of activities (e.g. Detailed Design -> Object Design)
- **Cutting**: Removing activities not need in the project (e.g. Maintenance not needed)
- **Ordering**: Defining the order the activities take place in (e.g. Testing before Implementation)

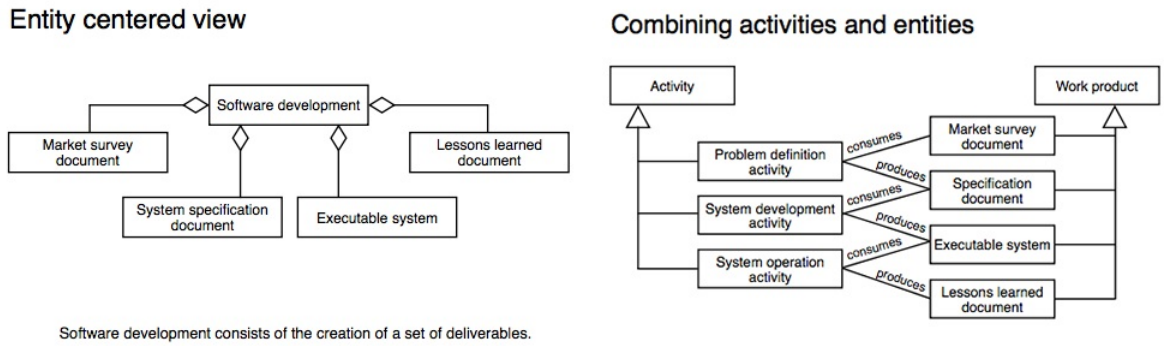
2.2 Modeling a Software Lifecycle

- **Functional Model of a Software Lifecycle**
  - Scenarios
  - User Stories
  - Use Cases
- **Structural model of a software lifecycle**
  - Object Identification
  - Class Diagrams
- **Dynamic model of a software lifecycle**
  - Sequence diagrams
  - State chart
  - Activity diagrams



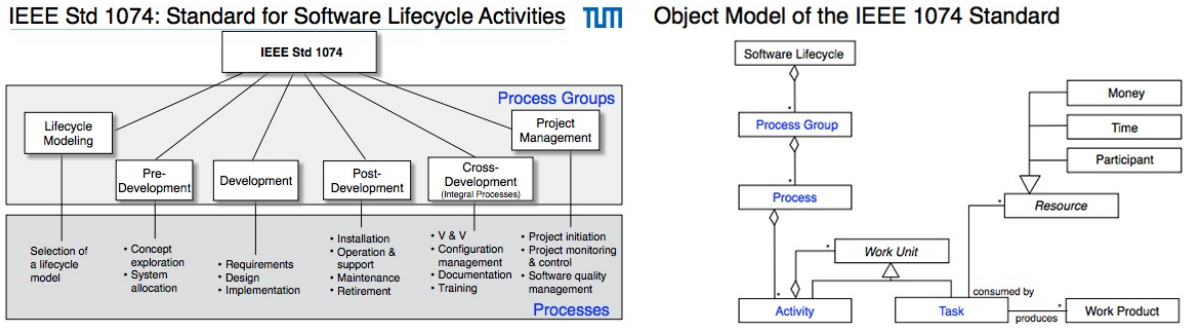
2.3 Major Views of Software Lifecycles

1) Activity Centered: Software development consists of a set of development activities.
2) Entity Centered: Software development consists of a set of deliverables



2.4 IEEE Std 1074: Standard for Software Lifecycle Activities

Process Group: consists of a set of processes (e.g. *Development)*
Process: consists of activities (e.g. *System Design*)
Activity: consists of sub-activities and tasks (e.g. *Persistent Data Managment*)



3. Overview of Software Lifecycle Models

Types of software lifecycle models

Sequential: Waterfall Model, V-Model (if changes are not to be expected)
Iterative: Spiral Model, V-Model XT, Unified Process (if changes are infrequent)
Agile: Extreme Programming, Kanban, Scrum (if changes are frequent)

The Waterfall Model

- Activity oriented
- Only once an activity is finished the next can start. (sequential)
- Assumption: software development can be scheduled as a step-by-step process that transforms user needs into code
- Once you are done with an activity, all issues covered in that activity are closed and cannot be reopened
- Activities:

- Concept Exploration
- System Allocation
- Requirements Elicitation
- Design
- Implementation
- Verification & Validation
- Installation
- Operation & Support

V-Model

- activity oriented
- sequential
- separation between development and testing
- mapping between development and testing activities
- Activities:

```
* Requirements Engineering - - - - - - - - Acceptance Testing
* - - - Requirements Analysis - - - - - - -System Testing
* - - - - - - - - System Design - - - - Integration Testing
* - - - - - - - - - Object Design - - Unit Testing
* - - - - - - - - - - - - - - Implementation
```

Validation: Assurance that a product, service, or system meets the needs of the customer and other identified stakeholders (often involves acceptance and suitability with **external** customers)
Verification: Evaluation whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition (often an **internal** process)

Spiral Model

- iterative
- 6 invariants of the spiral model
  - 1) Define artifacts concurrently
  - 2) Perform four basic activities in every cycle
  - 3) Risk determines level of effort
  - 4) Risk determines degree of details
  - 5) Use anchor point milestones
  - 6) Focus on the system and its lifecycle

V-Model XT

- successor of V-Model
- standard for German government contracts
- allows **tailoring** and **iterations**
- incorporates duties of the client
- 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

Unified Process

- Transition from engineering stage to production stages
- Each of the following stages defines
    - Objectives
    - Activities
    - Evaluation Criteria

```
[1] - - - - [2]  [1] Inception
 |           |   [2] Elaboration
 |           |   [3] Transition
[3] - - - - [4]  [4] Construction
```

\Extreme Programming (XP)

- initially released in 1987
- initial excitement, followed by lots of problems
- mostly performance problems

Kanban

- 4 Basic Principles
    - 1) Start with the existing process
    - 2) Agree to pursue incremental, evolutionary change
    - 3) Respect the current process,roles, responsibilities and titles
    - 4) Leadership at all levels

Scrum

- Management and control process
- Focus on early risk management
- Deliver working software, incrementally and empirically
- Simple framework for effective team collaboration

4. Summary

- A software lifecycle model (also software process model) represents a software lifecycle (concrete instance of a project)
- 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
- Different types of software lifecycle models:
  - **Sequential**: Waterfall Model, V-Model
  - **Iterative**: Spiral Model, V-Model XT, Unified Process
  - **Agile**: Extreme Programming, Kanban, Scrum

03 Scrum

17 May - 23 May

Slides: Moodle

1. Defined vs. empirical process control

How can we control software development with a process?

1. Through Organizational Maturity

- Repeatable Process
- Capability Maturity Model Integration (CMMI)

2. Through Agility

- Software development is empirical by nature
- It cannot be modeled with a defined process

Define vs Empirical Process Models

Defined Process: Planned, Follows strict rules, Avoids deviations
Empirical Process: Not entirely planned; rather 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 well defined inputs, the same outputs are generated every time
  - Precondition: All tasks and activities are well defined and provide **predictability** and **repeatability**
  - If the preconditions fail: surprises, loss of control, incomplete / wrong work products

Empirical Process Control Model

- not all pieces of work are completely understood; imperfectly defined process
- deviations, errors and failures are seen as opportunities that need to be investigated
- **Expects the unexptected:** control and risk management is exercised through frequent inspection
- Condition when to apply this model:
  - change is frequent and cannot be ignored
  - e.g. change of requirements, change of technology, change in the organization, change of people (?)

Scrum is based on an Empirical Process Control Model

- Definition (Rugby): A Scrum is a way to **restart the game after an interruption**
- Definition (Agile Processes): Scrum is a technique that **deals with interruptions** (change)
  - Manages and controls software and product development with rapidly changing requirements
  - Improves risk management by improved communication and cooperation and the delivery of product increments

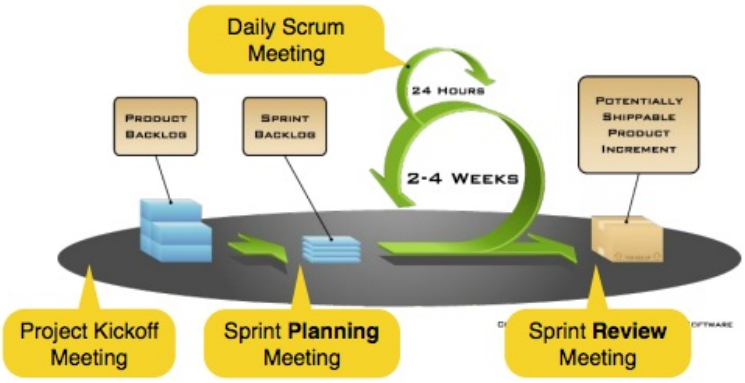
2. Scrum: An Example of Empirical Process Control

Scrum Artifacts

1. **Product Backlog:** List of requirements of the whole product
2. **Spring Backlog:** List of requirements for one iteration ("Sprint")
3. **Potentially Shippable Product Increment:** Release to the Product Owner that contains all results for the current sprint

Scrum Meetings

1. **Project Kickoff Meeting:** (at the start of a project)
   i. Create and prioritize Product Backlog
2. **Spring Planning Meeting:** (at the start of each spring)
   i. Create Sprint Backlog
   ii. Important: Development Team and Product Owner select items together
3. **Daily Scrum Meeting:** (everyday, about 15 minutes)
   i. Standup Meeting to share **status**, **impediments** and **promises**
4. **Sprint Review Meeting:** (at the end of each sprint)
   i. Demonstration of realized backlog items to the Product Owner (and other stakeholders)
   ii. Release and deliver the application (product increment)
   iii. Important: Product Owner gives feedback
   iv. Can be combines with the Sprint Planning Meeting for the next sprint.
5. **Sprint Retrospective:** (after each sprint)
   i. Opportunity to inspect the previous sprint and create a plan for improvements to be enacted during the next sprint



Scrum Roles

1. **Development Team:** self-organizing and cross-functional team, which realizes the product increment
2. **Scrum Master:** resolves impediments and is responsible for the process (e.g. visualization of progress with burndown chart)
3. **Product Owner:** defines the product and is responsible for the results

Sprint

- Time box during which the team creates a potentially shippable product increment.
- Typically 2-4 weeks long.
- Sprints have consistent duration throughout a project.
- A new sprint starts immedidately after the conclusion of a previous one.

3. Exercise: Modeling Scrum

// TODO: Model Scrum as activity diagram

```
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
```

4. Requirement Specification

Product Backlog

- Collection of items (user stories, scenarios) prioritized by the Product Owner
- Can be changed or reprioritized during projects
- Create on the basis of problem statement before the actual project starts (e.g. at Project Kickoff Meeting)

Priority

- describes the importance of a requirements for the software to be developed
- if a requirement has high value (e.g. core functionality), it has a high priority

  - **Critical:** (Prio 1) - candidates for the first development sprints / must be part of the first product increment
  - **Major:** (Prio 2) - must be realized within the project
  - **Minor:** (Prio 3) - desirable, if there is enough time
  - **Not Important:** (Prio 4) - might not be realized at all
- priorities can change during the project in Scrum
- Important: Prioritization is done by the Product Owner

Level of Difficulty

T-Shirt Sizes

- T-Shirt sizes represent the rough difficulty of an issue (how much effort / time the team needs to resolve the issue)
  - **Small (S), Medium (M), Large (L), Extra Large (XL)**
  - XL issues should be split up into smaller issues

Story Points

- The difficulty of backlog items is estimated in Story Points (1/2, 1, 2, 3,5 ...) (Fibonacci) using
  - Poker Planning
  - Team Estimation Game
  - other agile estimation techniques

4.1 User Stories

- a promise for conversations
- often written on a card
- users (**roles**) are the actual users of the system
- includes a sententce that describes **what the user does or needs**

User Story Template

> **As a** [ROLE]
> **I can** [FEATURE]
> **so that** [REASON]

Advantages of the template

- 'I'-phrasing makes it easy to identify with the role
- defines a structure and simplifies the prioritization process
- facilitates categorization into user roles

Properties of a good User Story (INVEST)

1. **Independent** - Avoid overlapping user stories
2. **Negotiable** - A user story is not a contract, but a basis for a discussion between Development Team and Product Owner

3. **Valuable** - for the user and the business. (Also **Vertical** - we plan and develop features not layers)
4. **Estimable** - the stories on the product backlog are the basis of our project plan
5. **Small** - too large user stories should be partitioned into smaller ones (complexity otherwise increases over-proportionally)
6. **Testable** - if a user story is not testable, it might not be of real value for the product (also, this implies realizability)

Acceptance Criteria

- conditions a software product must satisfy to be accepted by a user, customer or other stakeholder
- pre-established standards / requirements a product or project must meet
- set of statements, each with a clear pass/fail result that specify requirements applicable to the current stages of project integration
  - Functional
  - Non-Functional
- These requirements represent **conditions of satisfaction.** There is NO partial acceptance. Either a criterion is met or not.

4.2 Scenarios

- instance of a use case
- represents a concrete sequence of interactions between one or more actors and the system

Scenario Template

> **Scenario Name:**
> **Participating Actors:**
> **Flow of events:** [1] Actor Step Description
> [2] System Step Description
> ...
> [n-1] Actor Step Description
> [n] System Step Description
>
> **Entry Conditions:**
> **Exist Conditions:**
> **Invariants:**
> **Quality Requirements:** (also called non-functional requirements)

Relation between User Stories, Use Cases and Scenarios

- Scenarios are typically created during Anlysis
- Use Cases and Scenarios typically cover a larger scope and are more formal than User Stories
  - Scenarios and Use Cases have
    - Participating Actors

        - Preconditions
        - Event flow
        - Post Conditions
        - Special Quality requirements
    - Scenarios are instantiated use cases
- User Stories are usually created during requirements elicitation

5. Exercise Apply Scrum

Sprint Planning Meeting

- The Development Team estimates the difficulty for the most important items in the Product Backlog
- Development Team and Product Owner select Product Backlog items that can be realized in the Sprint
- The Development Team negotiates with the Product Owner how many items it can realize in the Sprint
- 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

Refinement of the Sprint Backlog

- The Development Team needs to discuss the concrete implementation to work on a backlog item
- The Development Team creates tasks to realize the backlog items (e.g. to match the acceptance criteria defined by the product owner)

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 (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) gives feedback and decides whether the items are realized completely
    - Realized items are ticked off in the Sprint Backlog
    - Unrealized items move back to the Product Backlog (= candidates for the next Sprint)
- The Product Owner can add new requirements to the Product Backlog or change existing ones

- 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 (“falsified”) or delivered (Product Owner decides)

Sprint Retrospective Meeting

- Scrum Master and Development Team meet to discuss how the previous Sprint worked out (Product Owner may participate)
- Different retrospective techniques exist (e.g. brainstorming what did (not) work out)
- Each team member is asked to identify specific things that the team should:
  - Start doing
  - Stop doing
  - Continue doing

6. Summary

Scrum as Methodology

- **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

Process Control

- **Defined process control** is entirely planned, follows strict rules and treats deviations as problems
- **Empirical process control** is not entirely planned and promotes frequent inspection and adaption
- Scrum is a popular agile method based on empirical process control that is used by many teams
- JIRA supports Scrum artifacts and activities

Scrum

- 3 Artifacts: Product Backlog, Sprint Backlog, Potentially Shippable Product Increment
- 4 Meetings: Project Kickoff, Sprint Planning, Daily Scrum, Sprint Review
- 3 Roles: Scrum Master, Product Owner, Developer

JIRA

- 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

04 Software Lifecycle Models II

17 May - 23 May

Slides: Moodle

1. Sequential Software Lifecycle Models

Properties of sequential models

Managers love sequential models

- nice milestones
- no need to look back (linear model)
- always one activity at one time
- easy to check progress during development (e.g. 90% coded, 20% tested)

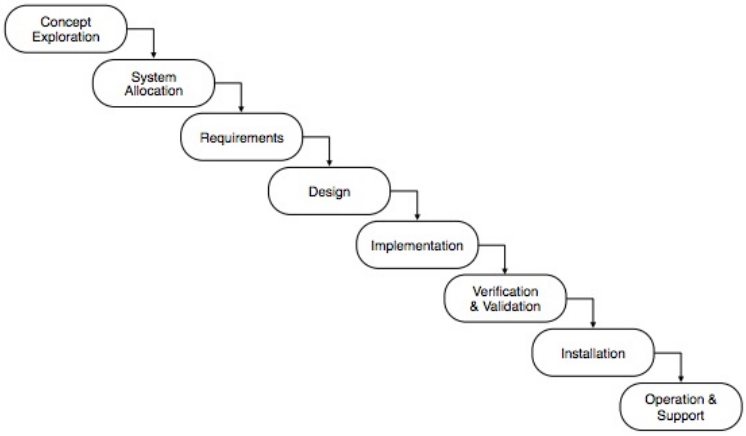
However, Software Development is non-linear

- **During Design:** problems with requirements are identified
- **During Implementation:** design and requirement problems are found
- **During Testing:** coding errors, design errors and requirement errors are found

Validation vs Verification

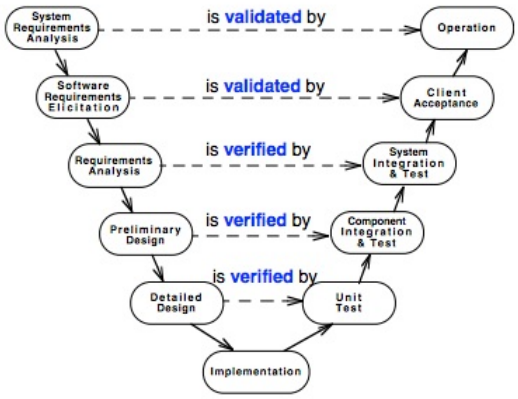
Validation	Verification
- Assurance that a product, service or system meets the need of the customer and other identified stakeholders	- Evaluation whether or not a product, service or system complies with a regulation, requirement or imposed condition
- Often involves acceptance and suitability with **external** customers	- Often an **internal** process
Informally: *Are you building the right thing?*	**Informally:** *Are you building it right?*

1.1 Waterfall Model



- An activity centered lifecycle model tat prescribes a sequential execution of activities
  - Assumption: Software Development can be scheduled as a linear, step-by-step process, transforming user needs to code
  - Goal: Never turn back, once an activity was completed
- Key Feature: **Verification Step** at the end of each activity
  - ensure that the activity does not introduce unwanted requirements or deletes mandatory ones
- Provides a simplistic view on Software development that measures progress by the number of tasks completed

1.2 V-Model



- The horizontal object flow denotes the information flow between activities on the same **abstraction level**
- **Higher Levels** of abstraction deal with the requirements in terms of elicitation and operation.
  - The Client Acceptance Activity validates the understanding of the user against the requirements
- The **middle part** of focuses on mapping the understanding of the problem into a software architecture
  - Component Integration and Test verifies functional components against the preliminary design
- **Lower Levels** focus on details such as the assembly and coding of software components

- Unit Test Activity verifies units against their description in the detailed design

2. Iterative Software Lifecycle Models

Iterative vs Incremental

Iterative	Incremental
- means to 're-do' or 're-work'	- means to 'add onto something'
- helps to improve product	- helps to improve process
Iterative Development: A strategy in which time is set aside to revise and improve parts of the system.	**Incremental Development:** A strategy where various parts of the system are developed at different times or rates and integrated as they are completed.

Increments can cause iterations

- a **product iteration** can result from **incremental addition** of functionality that had not been anticipated in previous iterations
- a **product iteration** can also result from **restructuring**

Limitations of Linear and Iterative Models

- Linear and iterative models do not deal well with frequent change
  - the Waterfall model assumes that once done with an activity, all issues are covered and will stay so
  - the Spiral model can deal with changes between activities, but not within and activity
- For more frequent changes use
  - Unified Process
  - V-Model XT
  - Agile Methods

Prototypes

Important: Don't use the term **rapid prototyping**

- confuses prototyping with rapid development
  - prototyping = technical issue
  - rapid development = management issue
- prototyping can go forever, if not restricted (time-boxed prototyping)

Types

1. **Illustrative Prototype**
   i. develop the UI with a storyboard
   ii. can be as simple as on a napkin or with dedicated tools
   iii. Good for first dialog with the client
2. **Functional Prototype**

   i. implement and deliver an operational system with minimal functionality
   ii. then add more functionality
   iii. good for incremental development
3. **Exploratory Prototype** ("Hack")
   i. implement parts of the system to learn more about the requirements
   ii. good for paradigm breaks

Vertical vs. Horizontal Prototype

- **Vertical Prototype:** Prototypes one feature (over several layers)
- **Horizontal Prototype:** Prototypes one layer (e.g. presentation layer)

2.1 Spiral Model

- Proposed by Boehm in 1987
- Consists of **9 iterations** (cycles)
  - [1] Concept of Operations
  - [2] Software Requirements
  - [3] Software Product Design
  - [4] Detailed Design
  - [5] Code
  - [6] Unit Test
  - [7] Integration Test
  - [8] Acceptance Test
  - [9] Implementation
- In each iteration **4 basic activities** must be applied
  - [1] Determine objectives, alternatives and constraints
  - [2] Identify and evaluate alternatives
  - [3] Identify and resolve risks
    - If resolved: Results are evaluated and the next iteration is planned
    - If not: immediate termination of project
  - [4] Develop a prototype and obtain approval for the next iteration

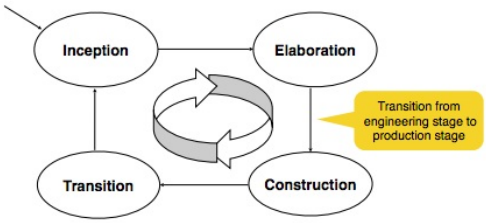
2.2 Unified Process

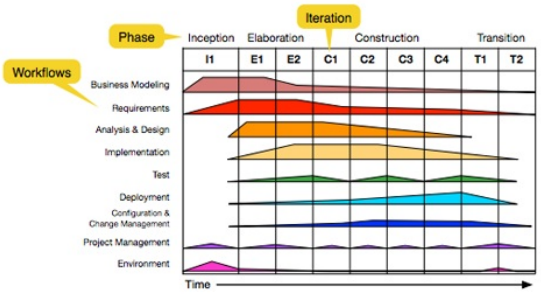
- An **iterative** and **incremental** lifecycle model built on the idea of **cycles** in the lifetime of a software
- Each cycle consists of **4 phases**
  - [1] Inception
  - [2] Elaboration
  - [3] Construction
  - [4] Transition
- Each phase can be iterated. During an iteration several **workflows** are performed in parallel
  - **Core Workflows:**

    - Business Modeling
    - Requirements
    - Analysis and Design
    - Implementation
    - Test
    - Deployment
  - **Supporting Workflows:**
    - Configuration & Change Management
    - Project Management
    - Environment

Phases of the Unified Process

The **2 phases** of the Unified Process are decomposed into 4 distinct phases





- **Engineering Stage:** (Inception and Elaboration)
  - less predictable and smaller teams
  - focusing on design and synthesis activities
- **Production Stage:** (Construction and Transition)
  - more predictable and larger teams
  - focusing on construction, testing and deployment activities

Phase vs. Iteration

- A **phase** creates a formal stake-holder approved version of artifacts
  - leads to "a major milestone"
  - phase-to-phase transitions are triggered by significant business decisions (not the completion of software development activities)
- An **iteration** creates an informal, internally controlled version of artifacts
  - leads to "a minor milestone"
  - iteration-to-iteration transitions are triggered by specific software development activities

2.3 V-Model XT

- successor of V-Model
- XT = extreme tailoring
- standard for German Government contracts

- incorporates duties of client
- Goals:
  - Minimize project risk
  - Improve and guarantee quality
  - Reduce costs
  - Improve communication
- Focus on products instead of activities (no defined chronology of activities)

General

- V-Model XT describes **WHO** has to do **WHAT WHEN** in a project
- Discriminates between different **Project Types** by
  - Role (customer, contractor_
  - Project topic (HW, SW, complex, embedded, integration)
- Each type has a **project execution strategy**
  - each stragey is a process model with milestones and decision points
  - V-Model XT = collection of different process models
- Task areas are described by **process components**
  - which define products, activities, roles and their relationships
  - some are mandatory (V-Model kernel), other subject to **tailoring**

Tailoring

- Select the applicable process components and a project execution strategy to create a project specific process.
- **Static Tailoring** occurs at project definition time (before the project starts)
  - project type and characteristics together suggest a set of process components
  - some choices are open (e.g. optional process components)
  - project specific adaptations may include
    - using different strategies for prototypes vs final development, etc
- **Dynamic Tailoring** may occur during the project execution
  - e.g. when HW functionality is to be realized, the component 'HW Development' will be dynamically included

Execution of a V-Model XT Project

1. Determine the project type and project characteristics
2. Choose a corresponding project execution strategy
3. Tailor the V-Model XT to your project
4. Repeat (Iterate)
   i. Identify all decision points to be reached in the next step
   ii. For each decision point:
      i. Identify all products to be generated for that decision point (direct or dependency)
      ii. Identify all activities needed for producing these products
      iii. Perform the activities and generate the products.

iii. Evaluate the decision points until the project is finished

3. Agile Software Lifecycle Models

Change influences choice of process model

PT: Project Time
TBC: Time Between Change
MTBC: Mean Time Between Change

Frequency of Change

- **Change rarely occurs:** (MTBC >> PT)
  - Sequential Model (Waterfall, 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 occuring during a phase may lead to the iteration of a previous phase or cancelation of the project
- **Change occurs frequently:** (MTBC << PT)
  - Agile Model (Scrum, Kanban, Extreme Programming)
  - Change during a phase can lead to reengineering of the requirements or design

3.1 Extreme Programming (XP)

- **Main Goals**
  - Avoid over-planning
  - Improve software quality
  - Improve responsiveness to changing customer requirements
- **Terminology:** Iteration, Deliverable, Release
- **5 Principles:**
  - rapid feedback
  - assume simplicity
  - incremental change
  - embrace change
  - quality work
- **4 Roles:**
  - Developer
  - Customer
  - Manager
  - Coach
- **12 Practices** how to approach the development process

Terminology

Iteration

- implements one or more user stories
  - time-boxed
  - finished on a fixed date
  - typical duration of 2-4 weeks
- variable number of features implemented

Deliverable

- the result of an iteration
  - coded, tested, potentially shippable
- Small addition of functionality

Release

- consists of several iteration that add related functionality
  - usually every 2-6 moths
- When should you release?
  - When the customer sees a big improvement over the last release

12 Practices grouped in 4 Areas

1. Rapid, Fine Feedback

- **[1] Test Driven Development**
- [2] On-site customer
- **[3] Pair Programming**

2. Continuous Process

- [4] Continuous Integration
- **[5] Refactoring**
- Short releases

3. Shared Understanding

- [7] The planning game
- [8] Simple Design
- [9] Metaphor
- [10] Collective Ownership
- [11] Coding standards

1. Developer Welfare

- [12] 40 hour week

Test Driven Development

- develop tests before you start coding
- write tests for
  - functional requirements
  - non-functional requirements
  - adherence to standards
- write only tests that could possibly fail

Refactoring

- = a change that ;eaves the system behavior unchanged, but enhances simplicity, flexibility, understandability and/or performance
- refactor only when the system requires you to do so
- run all tests after refactoring

Pair Programming

- All production code is written with 2 people looking at 1 machines
- 2 roles in each pair
  - **Driver:** Thinks tactically, worries about the implementation, writes tests and productions code
  - **Navigator:** Observer, thinks strategically, is the approach working? will a testcase fail?
- Pair Programming spreads knowledge

3.2 Kanban (covered in separate class)

3.3 Scrum (covered in separate class)

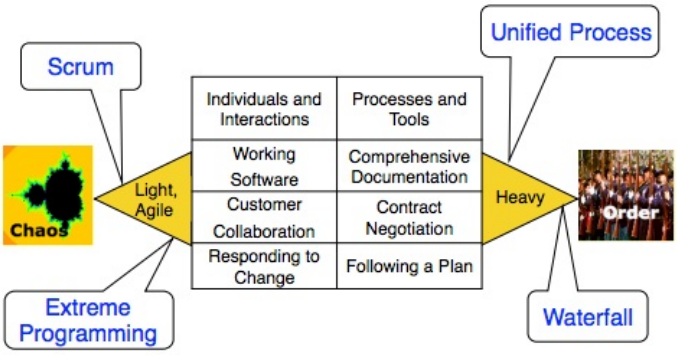
4. Summary

Limits of Empirical Process Control Models

= situations where agile methods might not be appropriate

- when not supported by the company culture
- a project too big for regular complete integration
- environments where is inherently takes a long time to get feedback
- where you cannot realistically test

Agile Manifesto: Dealing with Uncertainty, Complexity and Change


Scrum
Unified Process
Individuals and Interactions
Processes and Tools
Working Software
Comprehensive Documentation
Customer Collaboration
Contract Negotiation
Responding to Change
Following a Plan
Chaos
Light, Agile
Heavy
Order
Extreme Programming
Waterfall

05 Usability Management I

31 May - 6 June

Slides: Moodle

1. Motivation

1.1 Usability Disasters

Many Software Failures are due to Faults in the User Interface. Examples:

- German / Munich MVV ticket machines
- Questionably labeled elevator buttons
- Control rooms in nuclear power plants

1.2 Terminology

Usability

- measures how well a user can use the system functionality
- is measured in 5 categories
  - **[1] Learnability:** faster to learn => higher usability
  - **[2] Efficiency:** less steps to accomplish => higher usability
  - **[3] Memorability:** how quickly can a user reestablish proficiency
  - **[4] Errors:** amount of errors by the user, severity, recoverability
  - **[5] Satisfaction:** users come back more often, if the design / experience is pleasant

"The system is easy to use" - frequently misused term, often such systems are rather unusable
Unusability - the user has extreme difficulties to learn or use the system
User Experience - A person's perceptions and responses that result from the use and/or anticipated use of a product, system or service. (= all effectsof a product /service on the user, **before, during** and **after** the interaction)

(Good) User Interfaces are hard to design

Developer is not equal to the user

- Developers mostly communicate with other developers
- UI 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

UI development takes effort

- 50% of design, implementation and maintenance

Managers must be involved (usability management)

Usability Management

Usability is multidimensional

1. Learnability: Is the UI easy to learn?
2. Efficiency: Once learned, is it fast to use?
3. Memorability: Is it easy to remember what the user has learned?
4. Error Handling / Robustness: Are errors recoverable?
5. Satisfaction: Is the UI enjoyable to use?

2. Usability Heuristics (Nielsen)

2.1 Methods to reach good usability

Heuristics Evaluation

- a usability engineering method to find usability problems in user interface design

Usability Testing

- watching a user interact with the user interface of the system
- uses scenario-based testing
- involves 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 behavior

Difference: Usability Testing vs Heuristics Evaluation is similar to Walkthrough vs Inspection

2.2 Nielsen's 10 heuristics

Meet the Expectations

1) Match the real world
2) Consistency and standards
3) Help and documentation

The user is the boss

4) User control and freedom
5) Visibility of system status
6) Flexibility and efficiency

Handle Errors

7) Error prevention
8) Recognition, not recall
9) Error reporting, diagnosis and recovery

Keep it simple

10) Aesthetic and minimalist design

1. Match the real world

- Speak the user's language
- Don't limit user defined names
- Allow aliases in command language
- Use metaphores
- Natural Mapping

2. Consistency and standards

- Principle of least surprise
  - similar things should look and act similar
  - different things should look different
- Other properties
  - consistent size, location, color, wording, ...
- Command / Argument Order
  - Noun-Verb-Order vs Verb-Noun-Order
- Follow platform standards

3. Help and documentation

- Users don't read manuals
- BUT manuals and online help are vital (if user is frustrated)
- Help should be:
  - searchable
  - context sensitive
  - task oriented
  - concrete
  - short

4. User control and freedom

- Clearly mark exits
- Users should be able to explore interface (cancel / undo option)

- Users should not feel trapped by the interface
- All dialogs should have a cancel button

5. Visibility of system status

- Keep user informed about system state
- Response times
  - < 100 ms (instantaneous)
  - < 1s (user notices, but no feedback needed)
  - < 5s (display busy cursor)
  - > 5s (display progress bar)

6. Flexibility and efficiency

- Provide shortcuts for frequent operations
  - keyboard accelorators
  - command abbreviations
  - styles
  - bookmarks
  - history
  - ...

7. Error prevention

- Selection is less error prone than typing
- Disable illegal commands

8. Recognition, not recall

- Use menus, not command languages
- Use combo boxes, not text boxes
- Use generic (polymorphic) commands where possible (Open, Save, Copy, Paste)
- All needed information should be visible

9. Error reporting, diagnosis and recovery

- Be precise (restate user's input)
- Give constructive help
- Be polite and do not blame
- No 'fatal errors' or 'illegal commands'
- Hide technical details (stack trace) until requested

10. Aesthetic and minimalist design

- Good design is about simplicity
  - less is more (KISS)
- Good graphic design
  - few colors / fonts

    - follow color guidelines
    - align controls
- Use concise language

2.3 Heuristics Evaluation

- application of Nielsen's 10 heuristics
- is an **inspection method**
    - different from user testing
- Performed by a usability expert
- Basic steps:
    - 1) Evaluator inspects the interface thoroughly
    - 2) Evaluator compare the user interface against Nielsen's 10 heuristics
    - 3) Evaluator provides a list of usability problems

Good Heuristic Evaluation

- Justify every problem with a heuristic
- List every problem
- Go through the interface at least twice
- Don't limit yourself to the 10 heuristics

Formal Evaluation Process

1. Training
    i. Meeting for Design Team and Evaluators
    ii. Introduce application, target users, domain, scenarios
2. Evaluation
    i. Evaluators works separately
    ii. Each of the generates written reports or oral comments
    iii. Focus on generating problems, NOT ranking them
    iv. 1-2h per evaluation
3. Severity rating
    i. Prioritize all found problems
    ii. Take the mean of all evaluators ranking
4. Debriefing
    i. Evaluators and UI designers discuss the results and brainstorm solutions

3. Usability Testing

Usability Testing Methods

- AB Testing
- Expert Interview
- Task (Scenario) based usability testing
- Think aloud protocol

- First use experience
- Eye / Mouse tracking
- User Questionnaires

How much testing?

- Testing is like sex: when it is good, it is very, very good; and when it is bad, it is better than nothing.
- Nielsen showed that
  - already **5 users** revealed **85%** of usability problems
  - **15 users** reveal almost all problems

4. Prototyping

*Prototyping is **externalizing** and **making concrete a design** idea for the purpose of evaluation.*

Why Prototyping?

- Instant gratification
- Tangibility (helps to understand a system early on)
- Improves poort communication
- Allows early decision making
- Mistakes can be found early

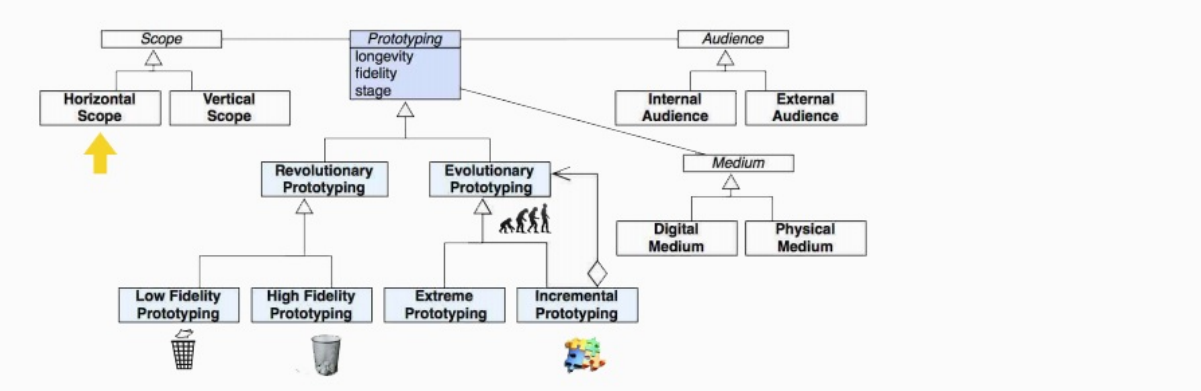
Failures are helpful

- Success in Engineering is defined by its failures (paradoxical approach to design)
  - better information comes from failing designs
  - failures draw more scrutiny and result in better solutions

Knowledge must be falsifiable

- There is no absolute truth; One can only build theories that are 'true' until someone falsifies it
- Falsification = the act of disproving a theory or hypothesis
- In software engineering any system (e.g. UI) is a model, thus a theory
  - models are often built to find counter examples
  - Testing = act of disproving a model
- Prototypes can be used to test

4.1 Types of Prototyping



Horizontal vs Vertical Prototypes

Horizontal Prototype	Vertical Protoype
Show wide range of features	Show small range of features (scenario / user story)
Horizontal Integration (bottom-up or top-down)	Full implementation of those features
Used in linear processes (no full implementation till the end)	Vertical Integration
	Used in agile processes

Integration Approaches during Prototyping

[1] Horizontal (traditional)
[2] Vertical (agile)
[3] UI-Centric (?? from both sides)

4.2 Evaluating Prototypes

- Heuristics Evaluation can already be used for the evaluation of prototypes
- Heuristics evaluation works on
  - sketches
  - paper prototypes
  - unstable prototypes
- "Missing element" problems are hard to find on sketches (=> look harder for them)

Tips for creating paper prototypes

- create as many as possible
- invest as little time as possible in the first iterations
- use test persons, which are in the target group of that software
- Don't limit yourself by thinking 'Is this even possible'
- Involve as many people as possible

5. Usability and Serious Games

- Serious Games are Games that involve additional goals - not only Fun.
- Serious Games increase motivation

Serious Game Categories

1. Gamification
2. Advergames
3. Learning-Games
4. Healthcare Games
5. Simulations

How to focus on serious games in your project

- Follow these heuristics
  - Gamification (ranking, badges)
  - Game development
  - 100 Qualities of Games
    - from The Art of Game Design: A Book of Lenses by Jesse Schell, 2012

6. Summary

General

- Usability is important
- Heuristics (e.g. Nielsen's 10 heuristics) help to evaluate / test the usability of a system

Benefits of Prototyping

- allow to get early feedback -> saves time while developing
- easier to understand the texts or diagrams -> easier communication
- improve usability

Benefits of Serious Games

- increase motivation
- focus on fun

Usability

- measures how well a user can use the system functionality
- is measure in 5 categories
  - Learnability
  - Efficiency
  - Memorability
  - Errors
  - Satisfaction

- Meet expectations, The User is the Boss, Handle Errors, KISS

How to reach good usability

- **Follow Usability Heuristics:** e.g. Nielsen's 10 Usability Heuristics
- **Usability Testing:** Test your ideas
- **Prototyping:** Identify problems early
- **Serious Games:** Focus on fun and motivation
- **Human Capabilities:** Model of the User (Newell)

06 Usability Management II

31 May - 6 June

Slides: Moodle

1. Prototyping

Revolutionary vs. Evolutionary Prototyping

Revolutionary Prototyping	Evolutionary Prototyping
=> get user experience with a **throwaway** version	=> used as the **basis** for the implementation
Advantage: Can be developed in a short amount of time	**Advantage:** Short time to market
Disadvantage: Users may have to accept that features in the prototype are too expensive to implement	**Disadvantage:** Can be used only if the target system can be constructed in a prototype

Low Fidelity vs High Fidelity Prototyping

	Advantages	Disadvantages
Low Fidelity		
	- Easy to Produce	- Mostly not reused
	- More Feedback	- Important details are ignored
	- No design decisione	
High Fidelity		
	- More realistic	- May cause much effort
	- More detailed problems can be	- Less feedback
	- More impressive	- High expectations (especially with interactive prototypes)

Risk of Prototyping

- Developer may become attached to the prototype
- Excessive development time of the prototype

- Customer might not appreciate how much work must be done to turn a prototype into a fully functional system
- User may confuse prototype with finished system
- Negative feelings towards the software, if the prototype has problems
- Cost of implementing a prototype

Managing Expectations

- Non-Programmers will use the state of the UI as a proxy for estimating how far program development is
- 90% worse UI => user will thing the program is 90% worse
- 100% beautiful UI => user will think program is almost finished

Storyboarding

- Modeling the UI as a Finite Automation
- Makes it possible to navigate through the UI with the users
- Often models can be used to generate code
- Allows to **combine evolutionary prototyping with low-fidelity prototyping**

2. Human Capabilities

Model of the User as a cognitive machine

- **Perceptual Processor**: Eyes, Ears
- **Motor Processors**: Muscles
- **Cognitive** Processor: Brain
- Long-term / working memory

Processor Cycle Times

- **Cycle Time** = time to accept an input an produce an output
- Eyes / Ears [50ms - 200ms]
- Muscles [25ms - 170ms]
- Brain [30ms - 100ms]

Fitt's Law

- Specifies how fast you can move your hand (mouse) to a target of certain size at a certain distance without overshooting it.
- Time = a + b * log (Distance / TargetSize + 1)

Consequences

- Targets at the screen edge are easy to hit (no correction cycle needed)
- Hierarchical menus are hard to hit (many correction cycles)
- Pie menus are faster than popup menus (each menu item is the same distance away from the pointer)

Model of the Human Memory

Memory Attributes

- Encoding: type of things stored
- Size: number of things stored
- Decay Time: How long does the memory last?

Long Term Memory

- Large Capacity
- Little decay
- Elaborative practice moves chunks from short term memory to long term memory by making connections with other chunks

Short Term Memory

- Capacity: 7 ± 2 chunks
- Fast Access Time: ~70ms
- Fast Decay: 7*[5s - 226s]
- depends on number of chunks
- distraction destroys short term memory
- Interference: A distraction where different stimuli activate conflicting chunks
- these chunks are harder to retain in short term memory
- example: color names, written in a different color

Noise in communication

- Explanation for *interference example with colors*
- at school we learned that words are more important than color
- saying the color of each word slows us down
- making a choice between conflicting options slows us down
- When projects become complex many choices need to be made
- Preferences and practices learned may unconsciously make decisions, when the requirements of a technology or system are not completely understood

3. Summary

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

Prototyping Risks

- Wrong expectations about the progress of non-programmer
- User confusion between prototype and finished system

Other

- **Human capabilities** are limited: users rely on clues (e.g. natural mapping, affordances) in order to learn from the design model
- **Fitt's Law**: how fast you can move your hand to a target of a certain size at a certain distance
- Natural Mapping and Affordances are clues that help us to design user interfaces

07 Software Configuration Management I

7 June - 13 June

Slides: Moodle

1. Motivation

Why Configuration Management?

- Multiple people work on artifacts that are changing
- Mote than one version of the artifacts 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 the system

What is Configuration Management

- Definition of **Software Configuration Management**
  - a set of management disciplines within a software engineering process to develop a baseline
  - baseline = a work product that can be changed inly through a change control procedure
- IEEE 828-2012: Standard of Configuration Management in Systems and Software Engineering

Administering Software Configuration Management

- SCM is a **project function** with the goals to make technical and managerial activities more effective
- SCM can be administered in several ways
  - organization-wide
  - project-specific
  - distributed among all project members
  - mixture of the above

Configuration Management Roles

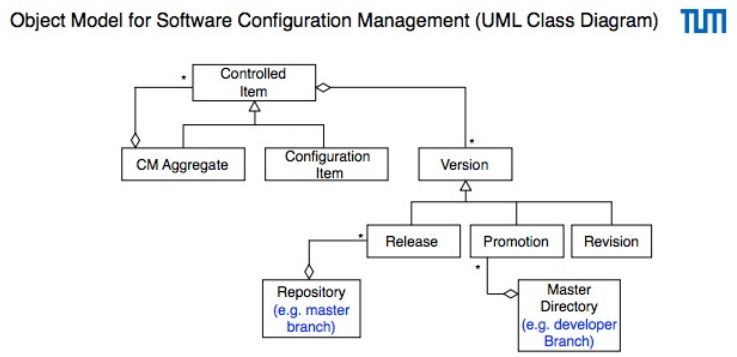
- **Configuration Manager**
  - Responsible for identifying configuration items

  - responsible for defining the procedures (workflows) for creating promotions / releases
- **Change Control Board Member**
  - responsible for approving or rejecting change requests
- **Developer**
  - creates promotions triggered by changes requests or the normal development activities
  - checks in changes and resolves conflicts
- **Auditor**
  - responsible for the selection and evaluation of promotions for release
  - responsible for the consistency and completeness of the release

Configuration Management Activities

1. **Configuration Item Identification** (Modeling the system as a set of evolving components)
2. **Promotion Management** (Creation of versions for developers)
3. **Build and Release Management** (Creation of versions for clients and users)
4. **Change Management** (Handling, Approving, Tracking of change requests)
5. **Branch Management** (Management of concurrent development)
6. **Variant Management** (Management of coexisting version) **(Not Covered)**

2. 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
  - Binaries
  - Documents
  - Configurations
- Hardware Configuration Items
  - CPUs

- Bus Speed Frequencies
- Sensors
- Actuators

Configuration Item Identification

- Not every entity need to be under configuration management control
- 2 Issues:
  - [1] **What**: Selection of configuration items
    - What should be under configuration control?
  - [2] **When**: When do you start to place entities under configuration control?
    - In early days, it was an activity
    - Today it should be a **project function** (from the beginning to the end of the project)

SCM Directories

- **Programmer's Directory** (IEEE 1042: Dynamic Library)
  - also called *working copy*
  - library for holding newly created or modified software entities
- **Master Directory** (IEEE 1042: Controlled Library)
  - manages the current baseline(s) and is for controlling changes made to them
  - changes must be authorized
- **Software Repository** (IEEE 1042: Static Library)
  - Archive for the various baselines released for general use
  - Copies of these baseline may be made available to requesting organizations

Version

- **Version:** An initial release or re-release of a configuration item associated with a complete compilation or recompilation of the item. Different versions have different functionality.
- **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. new commit in the version control system.
- **Revision:** Change to a version that correct only errors in the design / code, but does not affect the documented functionality.

Baseline

- A specification or product that has been formally reviewed and agreed to by the responsible management
- Thereafter is serves a the basis for further development and can be change **only** through change control procedures
- Example:
  - Baseline A: The API was completely defined. The bodies of the 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 are 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 Scheme of Baselines (Tagging)

- Many naming schemes exist
- Example: **A3 Digit Scheme**
    - [Major Release].[Minor Release].[Small Revision]

3. SCM Activities

3.1 Version Control (Promotion Management)

History of SCM Tools

- **RCS:** The first on the block (1975)
- **CVS** (Concurrent Version Control)
    - based on RCS
    - allowed working concurrently without locking
- **Perforce**
    - Repository server
    - allows to keep track of developers' activities
- **ClearCase**
    - Multiple servers, process modeling, policy check mechanisms
- **Subversion**
- **Git**

Version Control Systems (VCS)

- allow many software developers to collaboratively work on the configuration item of a project
- store different versions of configuration items in a commit history (and allow restoring those)
- allows developers to review how configuration items changed over time (commit history)
- stores revisions in a repository, where developers can check them out into their working copy
- Distributed Version Control Systems (DCVS) provide more flexibility and features

Version Control Architectures

Monolithic Architecture

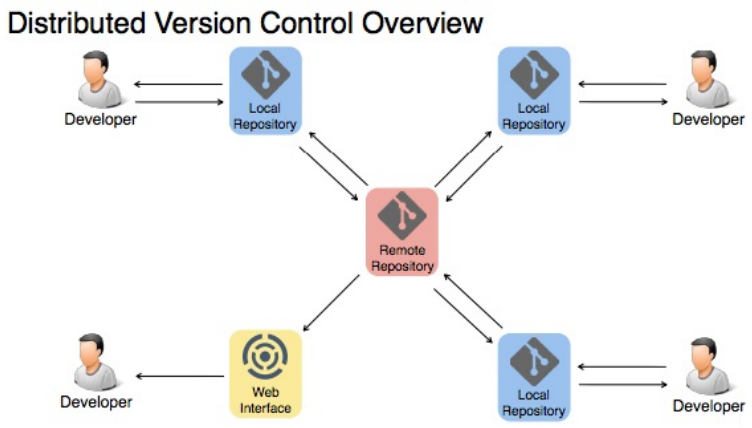
- Developers have a simple local database that keeps all changes to files under revision control
- Example: RCS
- Still distributed with many computers today

Repository Architecture

- A single (central) server contains all versioned files
- Developers can check out files from the server on their computer (programmer's directory), change them and check them back into the central server (master directory)
- Administrators have fine-grained control over who can do what
- **Problem:** Single point of failure at the central VCS server. Possibility of loosing all the versions and their history, if the server crashes.
- Example: Subversion

Peer-to-Peer Architecture

- Addresses the single point of failure problem
- Each "programmer's directory" fully mirrors the "master directory"
- Programmers can work offline and create versions (commits and branches)
- Not all versions are promoted to the "master directory" (they actively need to be pushed)
- If the server dies, the "master directory" can be fully restored from a mirror in the "programmer's directory"
- Example: Git

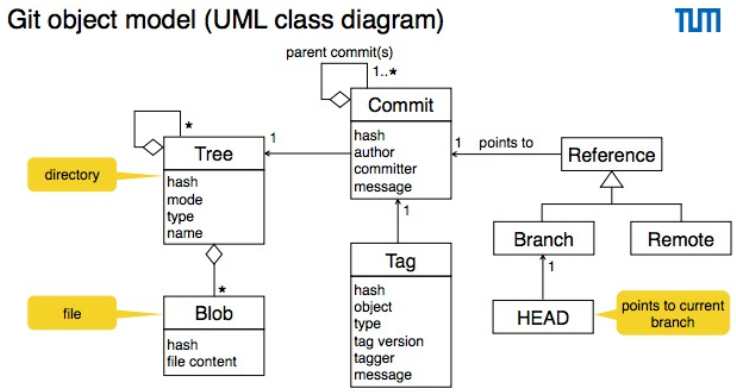


Comparison of DVCS and VCS

Advantages	Disadvantages
- Ability to work offline	- High learning curve
- Ability to work incrementally (small commits)	- scaling issues
- Ability for efficient context switches (lightweight branching)	- less administrative control
- Ability to do exploratory coding (lightweight branching)	

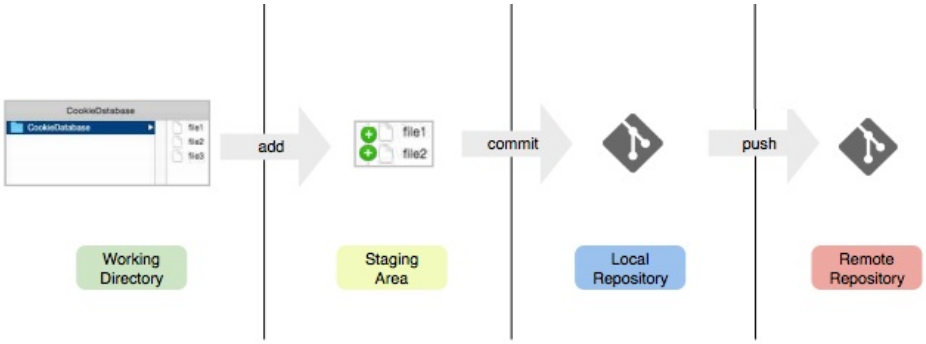
Git

- Open Source Project
- Supports lightweight local branching
- Differences to subversion
  - Multiple branches in each SCM directory
  - Branches are lightweight (file + history are not copied, but only referenced)



Git Commands

- **git add:** Add changed files to the staging area
- **git commit:** Commit selected changed files of the staging area to your local repository
- **git push:** Upload local commits to a remote repository
- **git pull:** (fetch & merge) Download and merge remote commits into your working copy
- **git clone:** (fetch & initial checkout) Clone a complete repository into a new working directory



Merge Conflicts

- Merge conflicts happen, if two persons work on the **same lines** in the **same files** at the **same time**
- 3 ways to resolve a merge conflict
  - [1] Take mine
  - [2] Take theirs
  - [3] Merge the overlapping changes manually

Best Practices: Distributed Version Control

1. Commit related changes
2. Commit and push often
3. Do not commit half done work
4. Test before you commit
5. Write meaningful and understandable commit messages
6. Do not use version control as a backup system
7. Keep your working copy of the repository up to date (regularly pull and push)
8. Use branches
9. Agree on a workflow
10. Do not change published (promoted) history

1. Commit related changes

- A commit should be a wrapper for related changes
- Two different bugs should be fixed in two different commits
- Small commits make it easier for team members to understand the changes

2. Commit and push often

- Keep commits small and concise
- Allows you to share code with other more often
- Easier for everyone to integrate their changes
- Having large commits probably leads to merge conflicts

3. Do not commit half done work

- Commit only if the code is completed
- Split feature's implementation into logical chunks and commit those early and often

4. Test before you commit

- Test thoroughly to make sure no side effects arise, before commiting (at least build the project)
- Even more important when pushing to the remote repository!

5. Write meaningful and understandable commit messages

- Begin message with a short summary of changes (~ 50 chars, reference to JIRA issue)
- Separate summary from body with newline
- The body of you message should provide answer to
  - What was the motivation for the change?
  - How does it differ from the previous implementation?
- Use imperative: ("Add XXX" instead of "Added XXX")

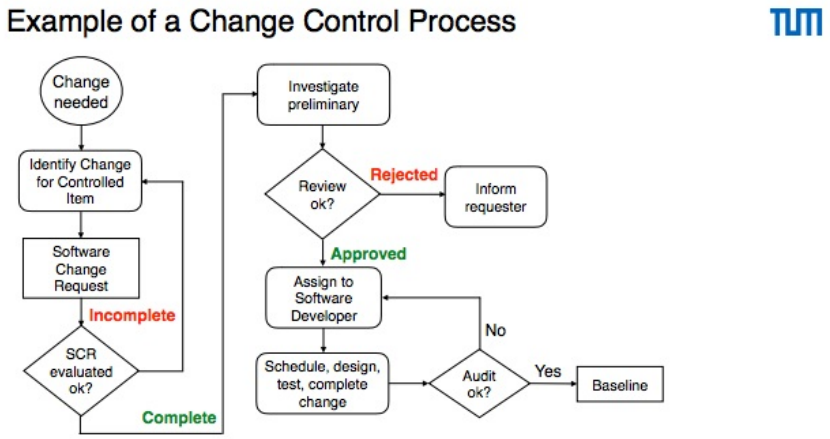
6. Do not use version control as a backup system

- Commits should be **semantical** (related changes)
- Don't just cram in files to back them up
- Do not commit binary files

- Resolve dependencies to other libraries automatically when building the source code
- Keep size of repository small (for performance reasons)

3.2 Change Management

- Change Management is the handling of **change requests**
- The general change management **process:**
  - [1] A change is requested
  - [2] The change request is evaluated against requirements and project constraints and reviewed by the **configuration control board**
  - [3] The change request is **approved** or **rejected**
  - [4] If it is approved, it is assigned to a developer, designed, implemented and tested
  - [5] The implemented change is audited to create another baseline



Change Request

- Specifies the procedures for requesting a change to a baselined configuration item and the information to be documented
  - Name(s) and version(s) of the configuration item(s) where a problem appears
  - Originator's name and address
  - Date of request
  - Indication of urgency
  - The need for the change
  - Description of the requested change

Change Policies

- The purpose of the change policy is to guarantee that each promotion or release conforms to commonly accepted criteria
- Examples for Change Policies
  - *No developer is allowed to promote source code that was compiled with warnings or errors.*
  - *No baseline can be released without having been beta-tested by at least 500 external testers.*

Change Management Activities and Responsibilities

1. **Software Configuration Control** (Managing a Change Request)
2. **Software Configuration Status Accounting**
3. **Software Configuration Auditing**

1. Software Configuration Control (Managing a Change Request)

- Define a change request form
- Define management procedure for
  - Identification of the need for a change request
  - Analysis and evaluation of a change request
  - Approval or disapproval of a change request
  - Implementation, Verification and Release of the change

2. Software Configuration Status Accounting

- Answers the following questions
  - What elements are to be tracked and reported for baselines and changes?
  - What types of status account reports are to be generated? What is their frequency?
  - How is information collected, stored and reported?
  - How is access to the configuration management status data controlled?

3. Software Configuration Auditing

- Identifies audits for the project.
- An **audit** determines for each configuration item, if it meets the required physical and functional characteristics.
  - Independent examination of a work product or a set of work products to assess compliance with specifications, standards, contractual agreements or other criteria
  - Audits are conducted according to a well-defined process consisting of various auditor roles and responsibilities
- The successful completion of an audit can be prerequisite for the establishment of the product baseline
- For each audit, the auditor has to define:
  - Objectives
  - Configuration Items under review
  - Schedule for the review
  - Procedures for conducting the review
  - Participants by job title
  - Required documentation
  - Procedure for recording deficiencies and how to correct them
  - Approval criteria

4. Summary

Software Configuration Management

- **Software Configuration Management**: A project function including 6 management activities within a software engineering process
  - **Promotions** are internal versions for other developers (e.g. made available through version control)
  - **Releases** are external versions for clients and users
- Change management controls how changes are handled in the process
- Distributed Version Control is state of the art and allows developers to collaboratively work on source code
  - benefits over centralized version control
- As developer and (technology) manager, you need to understand and use version control
- If you follow best practices, you work more efficiently and you avoid merge conflicts

Git

- Git is an example of a powerful and flexible distributed version control system which is widely used in practice
- You should be able to use the most important basic git commands
  - **Clone** a remote repository into your local file system
  - **Pull** changes from the remote into your local repository
  - **Commit** changes in your filesystem into your local repository
  - **Push** your local commits to the remote repository

08 Developing Winning Proposals

7 June - 13 June

Slides: Moodle

1. Motivation

1.1 Why Proposals?

Definition: Project

- Resources are brought together for a period of time, focused on a particular task.
  - e.g. development of a new product
- After the fulfillment of the tasks resources are disbanded and reassigned
- The management of project requires a special type of organisaton
  - to administer project resources effectively
  - to maintain clear accountability for the progress of the project

Why do we need proposals?

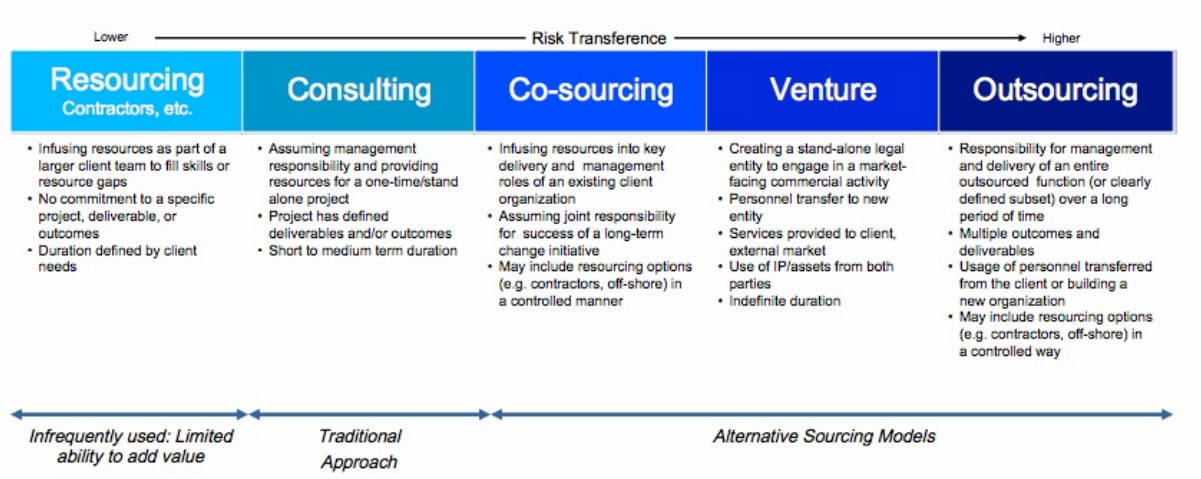
- Project based organizations are becoming more common
- Large organizations are split into separate companies
- Selling and winning is one part, understanding the scope the other (even more important)

1.2 What is a proposal?

Definition Proposal

- Legally binding, formal and comprehensive offer to a clients to solve a business problem by defining an appropriately tailored project
- (not an "offer of marriage")

Content and Effort to develop a proposal highly depends on the work proposed



For normal IT projects a wide variety of project types exist

- Strategic Work (IT Strategy)
- Conceptual Work
- Design and Implementation
- Project/Program Management

General Structure of a proposal

- Understanding the client's problem
- Describing the solution to the problem
- Work plan (tasks, time, budget, personnel) to solve the clients problem
- Prerequisites for the client
  - client time
  - client personnel involvement
  - project equipment
- Pricing
- Legal Issues (warranty, AGB's, etc)

Legal Issues are an important part of the proposal

- the proposal is legally binding
- highly important questions need to be answered (e.g. Werkvertrag vs. Dienstvertrag)
- involve a lawyer (if not, you can only loose)
- (not further covered here)

Pricing is the final step

- Historically projects were priced at "time & material" with a prearranged and negotiated area (between x and y) and the price for an hour/day was key
- More options exist now and are becoming more important
  - Fixed price for fixed scope ("all included")
  - Bonus for ahead of time delivery

- Value based billing

Sample

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

2. Proposal Process

Do you homework first!

- a structured approach is necessary even before biding on a project
- some clients/markets dictate the proposal process (public tenders)
- developing proposal is expensive => focus on the important ones

Step 1: Do you want to work for this client?

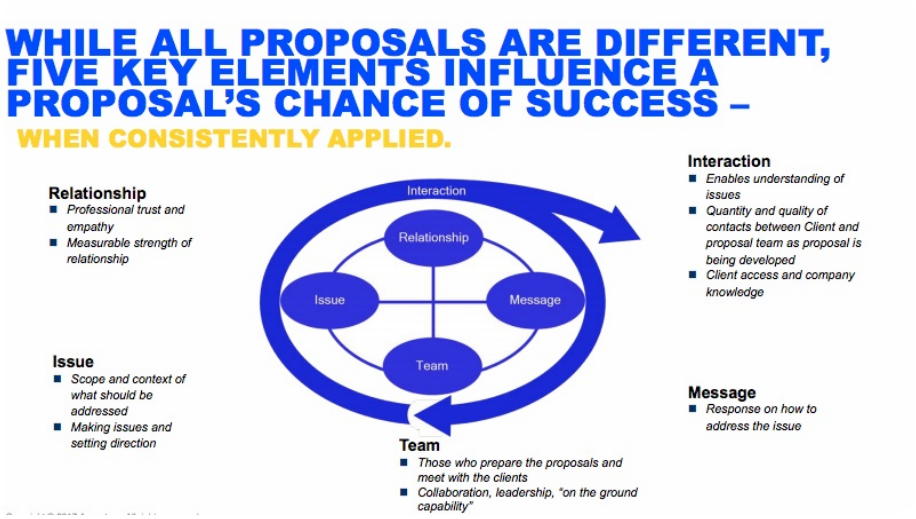
- Selected / Preferred Client (new / existing client)
- History of client towards working with external contractors
- Own history with the client
- Win-Win Situation?

Step 2: Do you want to bid for this project

- What is the competition?
- What are your strengths and weaknesses regarding the project?
- What is the estimated effort for developing the proposal?
- Does the project fit into your goals and strategy? (move to new area, reference project)
- Evaluate cost/risk proposition

3. Key Elements for a Successful Proposal

While all proposals are different five key elements influence a proposal's chance for success:



1. **Interaction**
    i. Enables understanding of issues
    ii. Quantity and quality of contacts between Client and proposal team as proposal is being developed
    iii. Client access and company knowledge
2. **Issue**
    i. Scope and context of what should be addressed
    ii. Making issues and setting direction
3. **Team**
    i. Those who prepare the proposals and meet with the clients
    ii. Collaboration, leadership, "on the ground capability"
4. **Message**
    i. Response on how to address the issue
5. **Relationship**
    i. Professional trust and empathy
    ii. Measurable strength of relationship

Recurring Relationships pay off

The focused pursuit of recurring relationships benefits both parties because it enables to understand client issues better and to ultimately add more value.

- Keep client's needs in the forefront
- Build peripheral vision of client issues while executing current engagement
- Reduce proposal efforts through better understanding of client issues
- Strengthen the five elements
- These factors all lead to more value added to the client

4. What makes a Winning Proposal

1. Understand your client's intention and have him/her understand you.

- People do not buy companies or ideas. They buy people.

2. Understand your client's needs and boundaries

- Assess decision parameters of the client thoroughly.
- A perfect 'full sized' solution will not be pursued, if it is way off the client's budget or the client wants to fix only part of the problem

3. Build the 'right' relationships

- Assess the buyer situation with a 'power map' of the client
- Who is the 'real buyer'? Who decides about the proposal? What is his / her motivation for the project?

4. Make sure that your team has the right skill-mix and be flexible to adjust

- The content of the proposal is key to winning a project
- You need and expert team to deliver quality content, relationships, legal structure, etc.
- If the focus of the proposal shifts to another functional area, make sure you include the necessary skills in your team

5. Learn from every proposal. Including failures

- Assess reasons for failures and lessons learned
- Share your knowledge and actively seek support from other and their experience
- Identify advantages in competitors

6. Be careful with your language

- Use professional language.
- Avoid overstating and exaggerating language

7. The proposal is only the first step of the project

- Promise only what you can deliver
- If you don't, you will not get another project with that client.
- It is harder to win trust than to lose it.

09 Kanban

14 June - 20 June

Slides: Moodle

1. Motivation

1.1 Challenges of Day-to-Day IT Project Work

Software Project Mananger

- Have a hard time setting clear priorities and making corresponding decisions
- They do not understand how their team works an why it 'does not work' the way they want it to.
- They are frustrated because the team does not manage to 'work according to plan'

Project Team

- Members are overwhelmed and stressed.
- The development takes longer than planned because the team members continuously have to justify their actions to management and are interrupted.
- The quality of the finished software is insufficient.
- They do not understand why the management makes such significant changes to the direction of the work and wants something different every day.

2. Origin: Toyota Production System

Motivation: Toyota

- Storage Capacity extremely expensive
- Production 'from stock' not financially feasible
- How can we optimize vehicle production so that we need as little storage capacity as possible?

Lean Production

- organize the flow of material similar to a supermarket
- consumer can pick and item, only then it is restocked
- lead to development of Toyota Production System

2.1 Toyota Production System (TPS)

Production only On-Demand

- Only produce what will be consumed

- 'Pull' mechanism forms the basis of Kanban
- Event-based (not clocked) system
- Challenge: Keep the pipeline filled.

Product Management using Cards

- Cards (Jap. Kanban)signal tasks to do
  - Handed over from station to station when a task is finished
  - Number of tasks **limited** for each station
- **Pull:** Only when a worker has nothing more to do, he will 'pull' the next task from the pipeline
- Challenge: Keep the pipeline going (everyone is busy)

Special Challenges

- **Lead Time:** Time between receiving a receipt and delivery of the product. (should be as short as possible)
- **Just in Time:** The production process should be designed as efficient and effective as possible in order to save costs. (storage costs, pipeline stall costs)
- **Optimization:** Minimize production effort while maximizing product quality.

2.1 Optimization of the TPS using Kaizen

- Kaizen is evolution, not revolution!
- Perfection is achieved not when there is nothing more to add, but when there Is nothing left to take away.

3 Steps

1. Recognize Potential (= eliminate redundancy)
2. Change the Process
3. Measurement and Analysis

Muda, Mura, Muri

- **Muda:** every activity / process that does not add value to the production (=> tailoring)
- **Mura:** waste resulting from inhomogeneity and inconsistency
- **Muri:** waste resulting from misused resources (machines / workers)

Muda

- Activities / Processes without added value
- Potential improvement in the following areas:
  - **Transport:** The movement of the product between different production plants and production steps
  - **Warehouse Stock:** Items currently WIP, material and finished products within the company
  - **Movement:** Physical movement of people and machines during operation
  - **Wait Time:** When people / machines are ready for operation, but have to wait

- **Overproduction:** Producing more than the customer ordered
- **Overworking:** Doing more than the customer needs
- **Defects:** Rejects and Defect Elimination during the production process

Mura

- Waste resulting from inhomogeneity and inconsistency
- Goal would be consistent, need-based capacity utilization

Muri

- Waste resulting from overload
- Potential improvement in the following areas:
  - Employees perform tasks they have not been trained to do
  - Non-ergonomic work spaces
  - Dirty and messy work spaces
  - Work instructions that are not clear
  - Insufficient tool support and kits
  - Insufficient maintenance and servicing
  - Tools and processes that are not allowed
  - Insufficient communication and insufficient feedback to the manager

3. Transfer: Kanban in IT

The **Development Process** in software development projects can be seen as the **Production Process**. Further, the **Requirements** represent the **Material** and the final **Software** the resulting **Product.**

3.1 Overview

Transparency: Management and Team share a common perspective

Management	Team
- Leadership instead of micromagement	- Considers the 'big picture', not just niches
- Makes the best possible decision	- **Kaizen:** continuous optimization of work methods / capacities in regard to cost, time and quality
- Communicates clear tasks prioritization	- Improvement without side effects, such as 'passive aggressiveness'
- Accepts the team's current work capacity and bases expectations on it	- Ability to make clear statements on capacity and performance

Core Principles of Kanban

1. Start with an existing process

2. Agree to pursue incremental, evolutionary change
3. Respect the current process, roles, responsibilities and titles
4. Leadership at all levels

3.2 Core Practices

1. Visualize the current workflow
2. Limit the work in progress (WIP)
3. Measure and optimize the flow
4. Make process policies explicit
5. Implement feedback mechanisms
6. Improve Collaboratively, Evolve Experimentally (Kaizen)

1. Visualize the current workflow

- A Kanban board should reflect what the team actually does
  - Backlog
  - Do
  - Develop (Ongoing / Done)
  - Deploy
  - Live

2. Limit the work in progress (WIP)

- Each column has a WIP limit
- The WIP Limit defines the allowed parallel work in progress in a specific process step
  - Development is ONE process step
  - It does not matter, whether a card is *done*, or *ongoing*, as long as it is in the process step, it counts towards the WIP limit.
- Only if the WIP limit is not yet reached new cards can be pulled in

Project Management is Process Controlling and Optimization

- **Flow Theory**
  - Cycle Time: Time needed for entire production process
- **Little's Law**
  - Throughput = WIP / Cycle Time
  - To increase the Throughput
    - **Reduce** WIP while **improving cycle time**
    - Traditional management would do it the other way around, which creates a downward spiral
- **Lean:** Problems MUST be solved
  - Implicit escalation enforces communication
  - **Slack Time** (= being blocked?) is wanted and used to solve problems, so they do not reoccur

3. Measure and optimize the flow

- **Lead Time:** Time from Request being made to Delivery
- **Cycle Time:** Time from Work Begins to Delivery

=> Kanban leads to predictability

4. Make process policies explicit

- **DOD**: Definition of Done
  - defines criteria, when a work item is ready for the next process step
  - agreed among team, customer and stakeholders
- DOD enables **Pull-Principle**

5. Implement feedback mechanisms

- Kanban Cadences (regular meetings for example)
  - [1] Standup Meeting
  - [2] Retrospective Meeting
  - [3] Replenishment Meeting
  - [4] Operations Reviews

6. Improve Collaboratively, Evolve Experimentally (Kaizen)

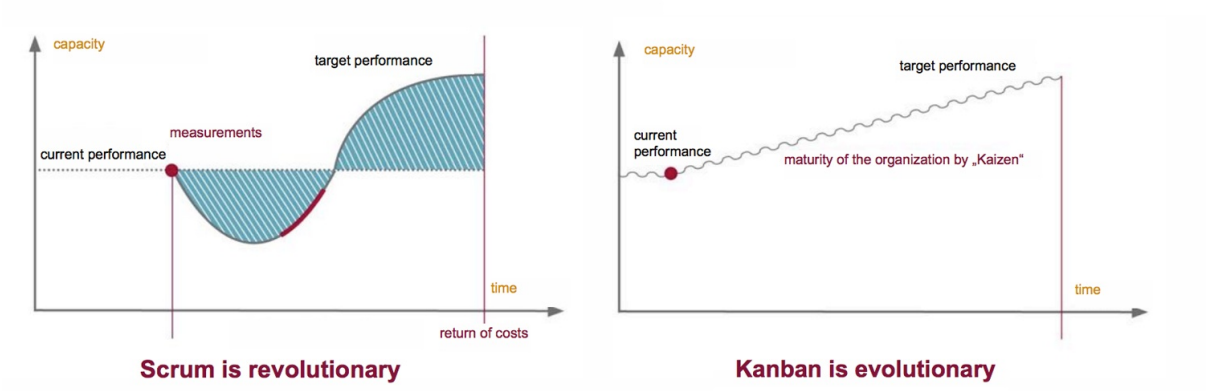
- e.g. Process Tailoring: Change the process steps under 'Development' to adhere to test-driven development

3.3 Kanban vs Scrum

Scrum is REVOLUTIONARY

Kanban is EVOLUTIONARY

- Small optimizations minimize the J-Curve effect



4. Summary

4.1 Properties of Kanaban

Kanban (method) stands for

- Transparency
- Optimization
- Personal responsibility

Kanban does not solve problems, but it makes them visible

Kanban begins with the current state and process of a system

- Current system development
- job titles and process is respected

Kanban is evolution NOT revolution

- Kaizen
- Kanban always considers the overall picture

Kanban requires to rethink traditional management

- Multitasking: => Stop Starting and Start Finishing
- Push without limits: => Pull with limits
- 100% capacity utilization: => 'Slack Time' is wanted

Kanban focuses on people, not mechanics

- Kanban is team sport

Kanban does not favor a specific management approach

Kanban can be combined with different approaches

- **Traditional:** Waterfall Model, V-Model XT, Spiral Model, Unified Process, ...
- **Agile:** Scrum => Scrum-ban

Kanban enables

- Incremental Process Tailoring (at runtime throughout the process)
- Process Change / Improvement Management

4.2 Principles and Core Practices of Kanban

4 Basic Principles

1. Start with an existing process
2. Agree to pursue incremental, evolutionary change
3. Respect the current process, roles, responsibilities and titles
4. Leadership at all levels

6 Core Practices

1. Visualize the current workflow
2. Limit the work in progress (WIP)
3. Measure and optimize the flow
4. Make process policies explicit
5. Implement feedback mechanisms
6. Improve Collaboratively, Evolve Experimentally (Kaizen)

10 Software Configuration Management II

21 June - 27 June

Slides: Moodle

1. Branch and Merge Management

1.1 Branch Management

- controls concurrent development
- defines rules for creating and merging (promoting) branches

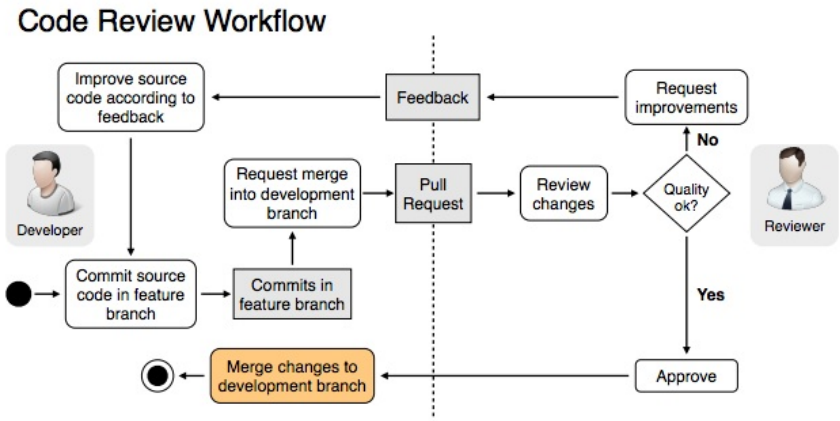
Example Branching Model

- **Master Branch:** External Releases (e.g. Product Increment)
  - external releases
- **Development Branch:** Promotions that can be used as internal releases
  - collection of promotions
  - candidates for releases
- **Feature Branches:** Incremental development and explorations
  - under control of one or more programmers

Merge Management with Pull Requests and Code Reviews

- Before changes in feature branches are merged (promoted) into the development branch, a reviewer reviews the changes
- The reviewer only approves changes, if the follow specified quality requirements

Code Review Workflow



Best practices for branch and merge management

- Use branches to control concurrent development and explorations
- Agree on a workflow (branching model)
- Map backlog items / requirements to feature branches (workflow automation)
- Only one person at once works with non-merge-able files
  - Communicate with your team
  - Minimize the working time on such files
- **Important:** Minimize feature branch lifetime
  - Split large requirements (e.g. user stories, scenarios) into smaller ones
  - Pull regularly from the development branch into the feature branch
  - Otherwise your branches might diverge heavily, resulting in serious, hard to resolve merge conflicts

1.2 Feature Toggles

- In practice, some feature branches are open for a long time, due to large requirements and lazy or overburdened reviewers
- Therefore some teams have different development models
  - Only one branch exists
  - Each commit is reviewed
  - Feature toggles are used to activate features once they are finished

2. Summary

- A branching model controls the concurrent development and defines rules when branches are created and merged
  - structure development according to backlog items
  - work on the same codebase with multiple developers
- Pull Requests facilitate to communicate directly on the changed code
  - Only reviewed code (high quality) will be merged to development (prevents broken window theory)
  - Reduced overhead for code reviewers (only changes need to be reviewed)
  - Inexperienced team members improve their skills (asynchronous pair programming)
- Branch and Merge Management are important management activities to structure concurrent development

11 Contracting

21 June - 27 June

Slides: Moodle

1. Motivation

1.1 Why is Contracting important?

Questions need to be answered

- What are expected deliverables?
- What is the schedule?
- What about the money?
- What are the responsibilities of the contractor and the client?
- What about change management?

Important!

- A contract defines the **basic requirements** of a project (deliverables, time, money)
- A contract defines the **basic rules of collaboration**
- However: It does not dictate how to deal with uncertainties (requirements, innovation, change)

1.2 Contracting in the Project Lifecycle

How does contracting work?

- Part of the project preparation (pre-development)
- Procurement, Acquisition and Bidding Procedures

Acquisition and Bidding

- Can be very formal
  - e.g. rigorous evaluation and selection processes in the public sector
- Don't underestimate the effort

Critical

- Price-Performance Ratio
- Good Estimations
- Good Calculations

Contracting: Interaction

Decision Gate

- Milestones and Quality Gate
- Requires completed work products to determine project progress
- A project plan is a sequence of decision gates

Client Project

- Get a system
- Often Waterfall like

Contractor Project

- Build a system
- Flexible approach

=> The formal contracting flow

Client		Contractor
1. Request for Proposals	=>	
	<= (Offer)	2. Submit Offer
3. Contract Awarded	<= (Contract) =>	3. Contract Awarded
4. Iteration scheduled		4. Iteration scheduled
5. Acceptance Completed	<= (Software) =>	5. Acceptance completed
	<= (Final Project Report)	6. Project Completed
7. Project Completed		

Contracting: Roles

Procurement, acquisition and bidding require 'project roles' (Client, Contractor). A contract defines responsibilities, deliverables, schedules, procedure for BOTH of them

- **Client** (also Customer, Purchaser, Contracting Enitity)
  - Define initial requirements
  - Request proposal(s)
  - Monitor (proposal) projects
  - Acceptance testing
- **Contractor** (also Supplier)
  - Submits offers (bidding)
  - Ships results (software, concrete deliverable, services, ...)

It gets more tricky, if it is an internal project **without** a contract

- Who is the client? Who is the contractor?

- How to define the agreement?
  - Informal Agreement
  - Formal Agreement
  - Cross-Unit-Collaboration

2. Influence Factors

Projects have different influence factors. These are also relevant for the contracting procedures.

Main Influence Facors

- Goals
- Business Case
- Initial esitimations
- Legal restrictions

Context Variables

- Every project has a variety of context variables, which
  - need to be considered in the contract
  - and have to be addressed properly, while leaving some flexibility
- Context variables and formal legal requirements need to be balanced in order to create a suitable environment for the project

Contract Content

- **Services and Responsibilities**
  - Functional Requirements
  - Deadlines (procedures for delays)
  - Quality Requirements (non-functional requirements)
  - Price and Payment Schedule
  - Acceptance
- **Warranty and Liability**
- **Provisions**
- **Copyrights**
- **Regulations, Standards, Compliance**

3. Basic Procedures

3.1 Overview

A project usually starts with a dialog and need preparation (regardless of acquisition strategy)

Project Acquisition Strategies

1. Pro-Active
2. Re-Active

3. Client request for an offer
4. Call for bids/submissions

Preparation in Practice

- Plan the offer's development as a project
  - [1] Prepare the Offer
  - [2] Develop / Submit the Offer
  - ... (System is developed)
  - [3] Acceptance of the System under Development
- A contract is **always** concluded based on the accepted offer.
- Sum of this makes contracting very demanding.

3.2 Prepare the Offer

Experience and Observation are important for the offer preparation. Particular procedure may vary from company to company.

Input: Project Idea
Output: Project Proposal (Draft)

Steps

1. Collect and complete information regarding the client
2. Evaluate business value
3. Conduct feasibility study
4. Develop offer and sales strategy
5. Estimate cost and effort
6. Check resource availability
7. Initial project planning
   i. Cost Plan
   ii. Resource Plan
   iii. Risk Analysis
   iv. Scheduling (initial)
8. Create proposal

3.3 Develop the Offer

Input: Project Proposal (Draft)
Output: Project Proposal

Steps

1. Build proposal development team
2. Develop proposal development strategy
3. Develop and continuously improve proposal parts

    i. Sales Parts
    ii. Economic Parts
    iii. Solution Parts
    iv. Contract Parts
4. Develop initial project plan
    i. Project Plan
    ii. Effort and Cost
    iii. Resources

3.3 Prepare Acceptance

The acceptance procedure should define

- **Contractor:** Shipping, including preparation
- **Client:** Acceptance Test, including
    - Who is authorized to accept?
    - Formal or silent acceptance?
    - Partial Delivery
    - Reporting

Important: Already in the contract the procedure for the system's acceptance have to be defined. Acceptance means always a **reversal of the burden of proof**.

- delivery dates
- acceptance procedures

Acceptance Tasks

- Check for **completeness**: “Was everything shipped as expected?”
    - Basis: the contract
    - Further artifacts: additional agreements, change requests, management decisions
- **Verification** of the project results: “Was the software developed right?”
- **Validation** of the project results: “Was the right software developed?”

Problems in the acceptance phase

- Errors and bugs
- Acceptance delays
- Acceptance refusal
- Acceptance with conditions

4. Legal Aspects

"A contract is really just a set of written playing rules"

Content, responsibility, trust, and risk

- How is a contract **structured**?
- What are the **basic rules** for delivering scope and invoicing revenue?
- How is **risk** and **reward** shared between customer and contractor?
- How are **changing requirements** handled?
- What **relationship model** between contractor and client is implemented?

Contract Types

Fixed Price	Time & Material
Subject: A system	**Subject:** Resources and time
Price: fixed	**Price:** based on effort
Progress Definition: Milestone based, payment schedule	**Progress Definition:** Feature based, not necessarily congruent with payment schedule
Disadvantage: Low flexibility regarding changes (changes request, changes are expensive)	**Advantage:** Open for change requests, as long as the client pays
Project Risk: Transferred to the contractor	**Project Risk:** Transferred to the client
Prerequisites: Fully specified system	**Prerequisite:** Requirements can be under-specified

Relationship Model: Fixed Price

Client

- negotiation and contracting procedures are well known
- client sticks to waterfall process
- client provides extra description of product (fully specified system)
- contract ensures that client gets product as specified (but not necessarily what they want)
- low budget reliability
- high cost risk
- **Risk:** max(system) for min(price)

Contractor

- realized project exactly as stated in the requirements specification
- has better perspective on project than customer
- scope and status can be determined based on requirements
- fixed price project are not meant for change
  - changes have impact on archtitecture / functionality
  - many contractors have adopted this as business model (risk compensation)
- **Risk:** min(System) for max(Price)

Relationship Model: Time & Material

Client

- Goal: maximise value and get a meaningful solutions
- Allocate resources as needed
- Responsible for project success by
  - providing feedback
  - taking part in the project
- The idea: one can exchange the contractor during the project (prevent vendor dependencies)
- **Risk:** Contractor might increase effort (and cost) unnecessarily
- **Risk:** Contractor has no duty to ship a product

Contractor

- Yield supplies as stated in contract (only requested resources are delivered)
- Invoice as many person-days as possible (to make the project more expensive)
- Make yourself irreplaceable
- **Risk:** Client might refuse the price increase

Relationship Model Behavior

- In projects parties have **interests**
- Interests are reflected in the behavior and interaction strategy
  - **competitive** (my win is your loss)
  - **cooperative** (win-win)
  - **indifferent** (I don't care - you lose)
  - **dependent** (heads: I win - tails: you lose)
- Contract should be signed with a **win-win** interaction strategy
  - fair contract
  - allows open atmosphere during project

5. Practices

Why do we need different process models (alternatives)?

- Significant increase of agile methods (absolute and relative)
- Relative growth of rich processes (e.g. V-Model XT)
- **Consequence:** The classic approaches have to evolve

Contracting Practices

Overview

Several practices exist to make contract more flexible to adopt agile methods.

1. Maximum Price
2. Change for Free
3. Exit Point / Money for Nothing

Note: All these practices have currently no clear legal basis. They are based on agreements that clients and contractors make. There are no legal definitions of how to apply them.

Overview

1. Maximum Price

Idea: Work according to **Time & Material** but set a limit.

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

2. Change for Free

Idea: Client is participating in the project (on-site client), give continuously feedback and has a learning curve

- Allow the client to add features
- Identify other features that can be removed / stalled
- Update the contract accordingly

3. Exit Points / Money for Nothing

Idea: Define an approach where a project can be stopped, but good performance is awarded.

- Define exit points (under which conditions to stop the project)
- Define payments to compensate, or award early termination

Example (Money for Nothing): For each saved month contractor gets another 10k. Example (Exit Point): After 2 sprints, client can decide whether he wants to do the entire project or terminate it.

4. Combined Practices

Idea: Combine **fixed price** and **time & material** into a flexible contract.

Example:

- Basic Instrument: fixed price contract (with maximum price)
- Work is done according to time & material
- Exit points are defined (cash flow related, exit conditions)
- Money for Nothing is implemented for good performance

Note:

- Focus on **awards and achievements**
- Can also contain **penalties**

The agile fixed price contract is considered a good way to balance legal requirements an agile methods

However

- Agility says: be open to change
- Change is critical, but may influence quality / functionality of system
- May require a formal change request if contract relevant.
- Generally, this will work as long as everything is fine. But prepare for problem by
  - ensuring **mature change management** is in place
  - having an cooperative project atmosphere (trust and respect)
  - defining the contract with (minimal) **stable requirements** and **exchangeable requirements**

How to handle failure?

As soon as lawyers start talking, the project is a 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

6. Summary

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 (defined and legally solid)

- Fixed Prices
- Time & Material

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

What are contracts for? (And for what not ...)

- Contracts **define rules** => It is not the goals to outrun the partner
- Contracts **need balance** => risk share
- Contracts **need to be fair** => all participating parties have rights and duties

- It is good to have a contract, but in projects ...

  - talk about the system, not paragraphs

  - an open atmosphere is better than a legal skirmish
  - you should name escalation authorities that are NOT lawyers
- **Remember:** A court case may give you justice, but no working software ...

12 Estimation and Scheduling

28 June - 4 July

Slides: Moodle

1. Motivation

- Nearly 2/3 of all projects significantly overrun their cost estimates
- 64% of features included in products are rarely or never used

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 an Estimation

- **Costs**
  - Personnel
  - Material
  - Extra costs
- **Scope**
  - Number of requirements
  - Complexity of requirements
- **Time**
  - Development time
  - Project duration
  - Dependencies
- **Infrastructure** (often forgotten)
  - Rooms, technical infrastructure, ...

Estimation Personnel Cost

- **Personnel Type:** team leader, applicant domain expert, analyst, designer, programmer, tester, ...
- **Cost Rate:** cost per person per day
  - [1] Single cost rate for all types (no differentiation necessary)

    - [2] Assign different cost rates to different personnel types based on experience, qualification, skills
- **Personnel Costs = Person Days * Cost Rate**

Estimating Development Time

- Development time is often estimated by the formula: **Duration = Effort / People**
    - **Problem:** Larger teams increase the communication complexity and decrease productivity
- Product Duration cannot be reduced arbitrarily by adding more people (Brooks Law)

2. Estimation Modeling

Basic Estimation Model

- **Input:** Parametric Data
    - Size & Project Data
    - Software Process
    - Insight / Experience
    - Historic Data
    - Meta Model of Software Process
- **Output:** Estimate
    - Effort
    - Schedule
    - Cost

Top Down and Bottom Up Estimation

2 common approaches for estimations:

1. Top-Down Approach

- Estimate the whole project
- Breakdown to different project phases and work products
- **Problem:** You did not break down the work before.

2. Bottom-Up Approach

- Start with effort estimates for tasks at the lowest possible level.
- Aggregate the estimates until top activities are reached
- Preferred if you have activities and tasks and knowledgable developers

3. Traditional Estimation Techniques

Methods for estimating cost and effort

All methods require **knowledge about the subject.**

- **Expert Estimations:** (Based on experience and domain knowledge)

    - T-Shirt Sizes
    - Planning Poker
    - Delphi
    - Assessment meeting
- **Algorithmic Estimations:** (Based on KPIs, formulas, metrics)
    - LOC (Lines Of Code)
    - COCOMO II
    - Aron
    - Function Points

Expert Estimation

= guess from experienced people (gut feeling)

- no better than the participants
- suitable for atypical projects
- justification of the result
- important when no detailed estimation can be done
- works best if the estimates are for short term items

Algorithmic Estimation

Lines of Code

- Traditional way for estimating application size
- Advantage: easy to do
- Disadvantages:
    - Focus on developers poit of view
    - No standard definition of "Lines of Code"
    - You get what you measure (reuse and refactoring is ignored)
- **The use of LOC metrics for productivity should be regarded a malpractice.**

COCOMO

- Constructive Cost Model
- Top-down approach to estimate cost, effort and schedule software projects based on size and complexity
- **Assumptions**
    - Derivability of effort by comparing finished project (database)
    - System requirements do not change during development
    - Exclusion of some efforts (administration, training, rollout, integration)
- **Cocomo II**
    - Also applicable for iterative lifecycle models (e.g. Spiral Model)
    - Additionally addresses experience, developer skills and distributed development
- **Problems**
    - Judgement required to determine the influencing factors and their values

- Experience showed deviation form actual effort by up to a factor of 4 (Cone of Uncertainty)
- Some important factors are NOT considered:
  - skills of team members, travel, environment, UI quality, overhead cost

Problems with Traditional Estimation Techniques

Cone of Uncertainty

- lessons learned from NASA projects
- at the beginning (initial concept, before requirements elicitation) of a project estimates have an uncertainty factor of 4

Focus on completion of activities and not delivery of features

- customers get no value from completed activities

Wrong focus in schedule reviews

- The reviewers look for overlooked activities not for overlooked features

When faced with overrun schedules, teams

- attempt to save time by reducing quality
- introduce change-control policies that constrain highly valuable changes in the software system

4. Agile Estimation Techniques

T-Shirt Sizes

- Estimations are particularly challenging for inexperienced developers
- Start with a simple model: e.g. T-Shirt sizes
- Map T-Shirt sizes to story points
  - S -> 1
  - M -> 3
  - L -> 8
  - XL -> Too big, split into smaller issues
- Once developers are familiar, move on to more fine grained methods (e..g Poker Planning)

Planning Poker

- Packages expert estimations into an enjoyable approach
- All participants are estimators, not just the M
- The estimation team should be < 10 people
- Estimates are arrived by **consensus building** (Do not average!)

Procedure

1. Read the acceptance criteria and discussion points on the Confluence page
2. Discuss how you would implement the user story

3. Individually estimate the user story
4. Simultaneously reveal the cards
5. Compare the points on the cards with each other, and discuss the differences
6. If necessary, play a second and third round Planning Poker
7. Agree on a final decision

Challenges

- based on expert opinion
- other teams may come up with completely different estmation
- estimations change with skill / experience
- teams estimates, while individual members are owners
- many people estimating might lead to long discussion
- lazy developers may overestimate tasks
- people tend to average out the point, which does not facilitate a discussion

Advantages

- Group Estimation = Wisdom of the Crowd
- The conversation / discussion following an estimate is a great way to pool important issues
- Story points can be used to determine the Sprint Velocity

Tips for Planning Poker

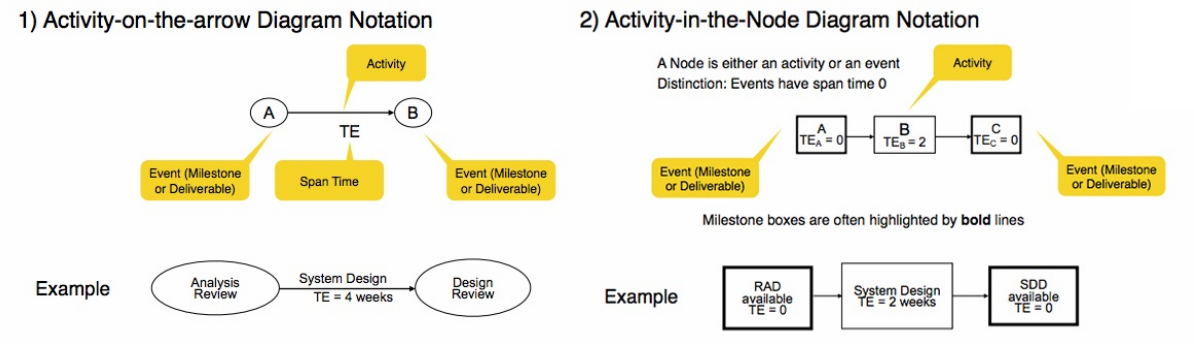
- **Keep discussions productive:** Set a limited amount of time for each user story discussion. This helps teams learn to estimate more rapidly within agile planning
- **Break out into smaller sessions:** Playing Planning Poker with a sub-set of the team is a good option if the number of user stories is too big
- **Choose the right time to play:** Consider playing it several times; once at the kick off, and once before each Sprint
- **Write down the reason for the awarded points:** If a user story receives too many points, make sure to note what were the Acceptance Criteria that led to that estimation
- **Consider the level of uncertainty:** Does the implementation of this user story depend on too many unknown factors? If yes, maybe award more points
- **Distribution of workload according to skill:** Developers who estimate few points are good candidates to implement the user story

5. 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 on event to the next
  - **Span Time:** The actual calendar time required to complete an activity
    - Relevant Parameters: Availability of resources, Parallelizability of Activity

- Dependency Diagrams are drawn connected graphs of nodes and arrows.
- Common notations are
  - 1) Activity on the arrow notation
  - 2) Activity in the node notation



What do we do with these diagrams?

1. Compute project duration
2. Determine activities that are critical to ensure timely delivery
3. Analyse the diagrame
   i. Find ways to **shorten project duration**
   ii. Find ways to do **activities in parallel**
4. 2 techniques used
   i. **Forward pass analysis** (determine critical path)
   ii. **Backward pass analysis** (determine slack time)

6. Critical Path and Slack Time Calculations

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

Non-Critical 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 tie that you can delay an activity and still finish your project in the shortest time possible

Analyzing Dependency Graphs

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

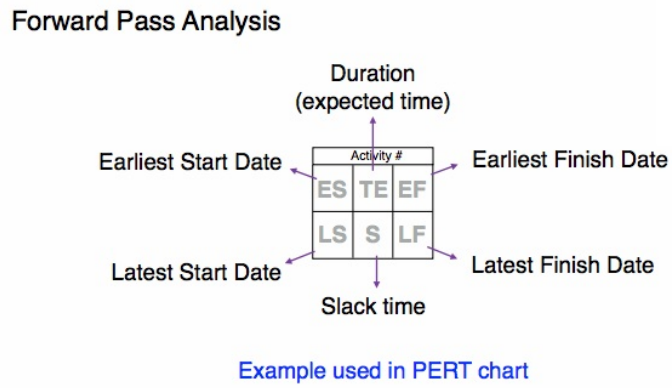
- **Latest start date (LS):** The latest date you can start an activity and still finish the project in the shortest time
- **Latest finish date (LF):** The latest date you can finish an activity and still finish the project in the shortest time

Forward Pass Analysis in PERT Chart

- Determination of **Critical Paths**
  - Compute the earliest start and finish dates for each activity
  - Start at the beginning of the project
  - Determine how fast you can complete the activities along each path until you reach the final project milestone

Backward Pass Analysis in PERT Chart

- Determination of **Slack Times**
  - Start at the end of your project
  - Figure out for each activity, how late it can start so that you still finish the project the earliest date possible



Procdure

1. Enter the **TE** (expected time) values for all activities
2. Starting from the first Event: For Each Event
   i. **ES** = max(**EF**) of previous node
   ii. **EF** = **ES + TE**
   iii. Continue for each subsequent event
   iv. Stop when
      i. encountering a node with missing dependencies
      ii. reaching the final event
3. Starting from the final Event (LF= EF): For each event
   i. **LF =** min(**LS**) of previous events
   ii. **LS = LF - TE**
4. Calculate Slack Time for each event
   i. **LS - ES**or **LF - EF**

5. Mark critical path(s)
    i. All nodes with Slack Time = 0 are on a critical path

Path Types in Dependency Graphs

Critical Path: Any path in a dependency diagram, in which all activities have 0 slack time
Non-critical Path: Any path with at least one activity with nonzero slack time
Overcritical Path: A path with at least one activity that has negative slack time. (such paths should be considered serious warnings for unreal estimates)

- Any dependecy graph with no fixed intermediate milestones has at least one critical path
- A project schedule with fixed intermediate milestone might not have a critical path

Types of Dependencies

- **Finish-to-Start:** Task B cannot be started before Task A finishes
- **Start-to-Start:** Task B cannot be started until Task A starts
- **Finish-to-Finish:** Task B cannot finish until Task A finishes
- **Start-to-Finish:** Task B cannot finish until Task A starts

Frequently used formats for schedules

Milestone View

- a table that lists milestones and the dates on which you plan to reach them
- good for introduction of project and high executive briefings

Activities View

- a table that lists activities and the dates on which you plan to start and end them
- good for documentation and during developer meetings

Gantt Chart View

- a graphical illustration on a timeline when each activity will start, be performed and end
- easy to read and understand
- **Person-Centered View:**
    - rows represent persons
    - determine people's workload
    - good for managing experienced teams
- **Activity-Centered View:**
    - rows represent activities

  - identify teams working together on the same task
  - good for managing beginners

Gantt Chart View with Milestones

- a Gantt Chart which contains milestones in addition to activities
- good for reviews

PERT

- PERT: *Program Evaluation and Review Technique*
- a graphical representation of task dependencies and times
- good for clear illustration of task dependencies
- Algorithm:
  - assign optimistic (O), pessimistic (P) and most likely (M) estimates for the span times of each activity
  - TE (Time Expected): **TE = (O + 4M + P) / 6**

7. Reasons for Schedule Overruns

Overview

1. Activities don't finish early
2. Lateness is passed down in the schedule
3. Activities are not independent
4. Multitasking causes more delays
5. Features are not developed by priority
6. Uncertainty is ignored
7. Estimates become commitments

1. Activities don't finish early

- Tendency that **one expands the scope** of the task to fill the estimated time
- Parkinson's Law: *Work expands so as to fill the time available for its completion.*

2. Lateness is passed down in the schedule

- Because traditional plans are activity based, they focus on the dependencies between activities
- Things start early only if EVERYTHING goes right. But the start late if anything goes wrong.
  - An **early** start of an activity does **require a combination of dependencies** to go well.
  - A **late** start of an activity does only **require one dependency** to be late.

3. Activities are not independent

- Activities are independent when the duration of one activity does not influence the duration of another activity
- In Software Development most activities are **inter-dependent**
- When activities are dependent, **variations in completion time do not balance out**

4. Multitasking causes more delays

2 Tasks: If you are block on Task A, work on Task B

- > 2 Tasks:
  - **Time spent switching between tasks become costly**
  - Tasks remain in progress longer
  - **Developer Overload:** loading every developer to 100% will congest the system

5. Features are not developed by priority

- Traditional planning assumes that all activities will be completed
  - If all work is completed the order does not matter
  - In reality the project team is often **scrambling to meet the schedule** and drops features
  - Features with high value might be dropped

6. Uncertainty is ignored

- Traditional planning approaches ignore uncertainty
- Better approach: Allow for short iterations
  - reduces uncertainty

7. Estimates become commitments

- Every estimate indicates a probability that the work will be completed in time
- Estimates are no commitment. There is a risk that they will not be met.

8. Schedule Heuristics

1. How to develop an initial project schedule

- Identify all your **activities**
- Identify intermediate and **final** dates that must be met
- Assign milestones to these dates
- Identify all activities and milestone **outside of your project** that may affect your projects schedule
- Identify "depend on" **relationships** between the activities
- Draw a dependency diagram for the activities and relationships
- Determine critical paths and slack times of non-critical paths

2. How to shorten the project duration

- Re-check the original span time estimates
  - Ask other experts to check the estimates
  - Has the **development environment** changed? (e.g. desktop vs mobile)
- Consider different strategies to perform these activities
  - Consider buying a work product (Buy vs Build)
  - Consider hiring an external contractor

- Hire more experienced personnel
  - **Brook's Law:** Adding manpower to a late software project makes it later.
  - Trade-Off: Experts work faster, but cost more
- Try to find **parallelizable** activities on the critical path
  - Continue coding while waiting for the results of a review
  - Risk activity, since portions of the work may have to be redone
- Develop an **entirely new strategy** to solve the problem

3. Mistakes when preparing schedules

Using Fudge Factors

- **Fudge Factor:**
  - the extra amount of time added to your best estimate to be "just safe"
- Don't use fudge factors!
  - If an activity takes 2 weeks and you add 50% fudge factor, chances are almost 0 that it will be done in less then 3 week (reason: Parkinson's Law)

The "Backing In" Mistake

- **Backing In:**
  - You start at the last milestone of the project and work your way back toward the starting milestone, while estimation durations will add up to the available time.
- Problems with Backing In:
  - You probably **miss activities** because your **focus is on meeting the time constraints** rather than identifying the required work.
  - Span times are based on what you "allow them to take", not on the time needed
  - The order of the activities may not be the most effective one.
- **Better:**
- Start estimating the required times and then try to shorten the project duration.

4. How to become a good software project manager

1.Don't assume anything

- Find out the facts
- use assumptions only as a last resort
- With every assumption risk is introduced

2. Communicate clearly with your people

- Being vague just increases the chances for misunderstanding

3. Acknowledge performance

- Tell you team members and peers when they are doing well

4. View your people as allies not as adversaries

- Focus on common goals not individual agendas
- Make people comfortable by encouraging brainstorming and creative thinking

5. Be a manager and technical leader

- Deal with people as well as with deliverables, processes, models and systems
- Create a sense of vision and excitement

9. Summary

Estimation is a basis for the decision to start, plan and manage a project

Estimating a project is a difficult project management function, however:

- Few software organisations have established a formal estimation process
- The estimation technique influences the result (Cone of Uncertainty)

Cost should not be estimated before the work is broken down into activities and tasks

- The dependency graph included the dependencies
- Schedule = Dependency Graph + time estimates

Critical Path Analysis

Many reasons for scheduling overruns

Scheduling Heuristics -> How to become a good manager

13 Build Management

5 July - 11 July

Slides: Moodle

1. SCM: Build Management

Requirements for Build Management

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

- regular builds from the master directory
- automated execution of tests
- email notifications
- determination of code metrics
- automated publishing of the application and test results

Tools for Build Managements

- Unix's Make
- Ant
- Maven

Activities in Build Management

The transition from source code to executable application contains many mechanical (usually boring) activities.

- Setting required paths and libraries
- Compiling Source Code
- Copying source files
- Setting file permissions
- Packaging the application

Executing these steps manually is time consuming and the chance of introducing failures is high.

Reasons for Continuous Integration

- **Risk 1:** The later the integrations occurs in a project, the bigger the risk that unexpected failures occur.
- **Risk 2:** The higher the complexity of the software system, the more difficult is the integration of its components.
- Continues Integration addresses these risks **by integrating and testing as early and frequently**

as possible.

2. Overview: Testing and Continuous Integration

Motivation

- Faults are everywhere

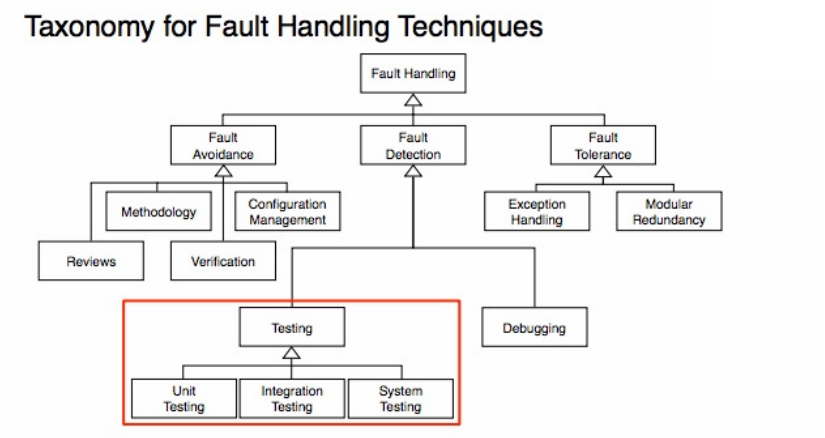
Terminology

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

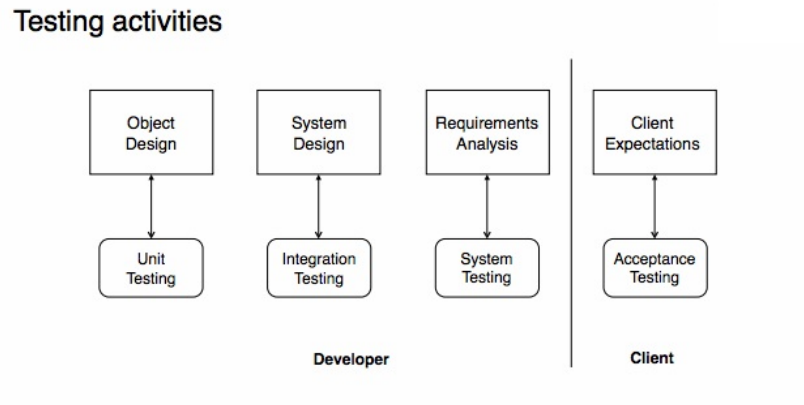
Fault Handling Techniques

- **Fault Avoidance**
  - use methodology to reduce complexity
  - use software configuration management to prevent inconsistency
  - apply verification to prevent algorithmic faults
  - use review 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 the state and unusual behavior (used during debugging)
- **Fault Tolerance**
  - Exception Handling, modular redundancy

Taxonomy of Fault Handling Techniques



Testing Activities



Types of Testing

- **Unit Testing:**
  - individual components (class or subsystem) are tested
  - carried out by developers
  - **Goal:** confirm that the component or subsystem is correct and carries out the intended functionality
- **Integration Testing:**
  - Groups of subsystems (collection of subsystems) and eventually the entire system are tested
  - carried out by developers
  - **Goal:** test the interfaces between the subsystems
- **System Testing**
  - the entire system is tested
  - carried out by developers
  - **Goal:** determine whether the system meets the requirements (functional and non-functional)
- **Acceptance Testing**
  - evaluates the system delivered by developers
  - carried out by the client and may involve executing typical transactions on site
  - **Goal:** demonstrate that the system meets the requirements and is ready to use

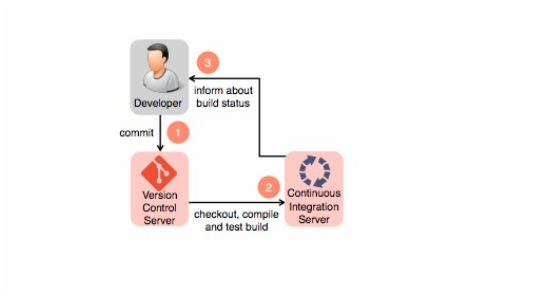
Acceptance Testing

- **Goal:** demonstrate that the system is ready for operational use
  - many tests can be taken from integration and system testing
  - choice of tests is made by the client
  - executed by the client, not the developer
- **Alpha Test:**
  - client uses the software at the developer's environment
  - software usage in a controlled setting with the developer ready to fix bugs
- **Beta Test:**
  - conducted in the client's environment (developer not present)
  - software gets realistic workout in target environment

Definition: Continuous Integration

A software development technique where team members integrate their work frequently. Usually each person integrates at least daily, leading to multiple integrations each day.

- Each interaction is verified by an automated build, which includes the execution of tests (regression tests) to detect errors as quickly as possible.
- **Advantages:**
  - There is always an executable version of the system
  - Developers and managers have a goof overview of the project status
  - Automatic **regression testing**
- **Workflow:**



Regression Testing

- **Goal:** verify that software previously developed and tested still performs correctly even after it was changed or interfaced with other software.
  - **Benefit:** errors in the source code are found immediately after a change is introduced
  - **Drawback:** can be very costly to execute a large test suite after each change
- Techniques:
  - [1] re-test all
  - [2] regression test selection (e.g. dependency analysis)
  - [3] test case priorization
- When to execute test cases:
  - after each change
  - nightly
  - weekly

Management Issues: Continuous Integration

A project manager needs to answer the following questions:

- How long does the integration and execution of test cases take?
- Can you only execute the test cases related to the change?
- How often should you do it? (each commit, nightly, weekly)
- Do you operate you own server or build in the Cloud?
- How many build agents do you operate?

3. Summary

- Fault avoidance and detection are important techniques to maintain the quality of your source

code and to prevent bugs.

- Make sure to write test cases
- ... and to set up continuous integration
- Advantages of continuous integration
  - you always have an integrated build of your software
  - you can find failing test cases immediately
- Bamboo
  - Lets you setup CI easily
  - Nicely integrates with JIRA and Bitbucket
  - The are also other tools ...

14 Global Project Management

5 July - 11 July

Slides: Moodle

1. What and How

What is special about global projects?

1. Distance
2. Time Zones
3. Culture

When does global context occur?

- **Within an organisation:** Global companies have different roles and/or system ownership locations around the world
- **Outsourced delivery model:** onsite, onshore, nearshore, offshor
- **More than one party involved:** implementation partner, hosting partner, cloud delivery, joint venture, ...

2. Communication

- Crucial for successful project deliveries and product development
  - Impossible to deliver value without communication
  - Impossible to utilize team capacity without communication
  - **Active communication will make any project successful**
- Miscommunication is a major reason for failed projects, cost overruns, quality problems, project delays
- 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 Context and Mitigation

- **Who to contact** - roster (dt. Namensliste)

- **Difficulty of initiating contact** - documentation communication structure
- **Language** - explain back to each other
- **Ineffective communication** - be aware in communication strategy
- **Lack of informal communication** - build relationships

What makes good communication?

- Awareness and respect to culture and workload
- Adapting messages to recipient
- Identifying where communication needs to be improved (find out why people are not communicating)
- Communication needs to add value for the recipient

Find the right tool to communicate

- Slack
- Trello
- Skype
- Jira
- Hangouts

Travel

- Invest in meeting face to face
  - Have a satellite in the other location channel things
  - When budgeting the project always negotiate for travel budget

Practical Tips

- 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

Lessons Learned in different roles

Context become more and more international. We are citizens of the world.

- **Requirement Lead**
  - Communicate clearly to developers
  - Ask them to explain back
  - Follow progress
- **Team Lead**
  - Encourage team to find communication hubs

  - Motivation of teams crucial
- **Solution Architect**
  - Initiate and fuel collaboration between teams for solution design & code structure
- **Project Manager**
  - Ask for honest input about possibility and feasibility of decisions/asks

3. Product Development in a Global Context

Differences in Product Development vs Projects

Product Development

- done with long term organization and with no decided end date
- important to invest in long term productivity and satisfaction
- adds dependencies by size and product lifecycle
- IT architecture is important
- more mechanisms to handle dependencies needed

Projects

- possible to create interimistic teams optimized for short term delivery
  - high investment – high energy
  - can more easily free themselves from patterns and habits and focus on the project goal

Architecture and Infrastructure for Global Context

- Think smart when “Slicing the Elephant”
  - split the responsibility for parts of the product in a way that minimizes dependencies
- If possible, choose an architecture that enables distributed teams
  - Continuous integration to avoid planning dependencies
  - Keep product dependencies to a minimum
- Data center location -> performance and data protection aspects

4. How does Agile Software Development Work in a Global Setup?

Agile Cornerstone	Challenge in Global Setup	Mitigation
Enable People	Dependencies and teams on other locations, feeling distance and being just one in the crows	Focus on visualization of teams and their responsibility, competence exchange
Close Collaboration	Teams become isolated, hard to see the full picture	Demos where teams share what they delivered each sprint
Team Autonomy	The later in the product lifecycle the more dependencies, hard to slice the elephant	Optimize teams based on autonomy, add collaboration forums to enable autonomy

Continuous Improvement	Dependencies and forced timeline leaves little room for improvements	Continuous improvement epic in the backlog
Innovation	Hard to find time for innovation	Schedule time for innovation between sprints

Spotify's way of scaling

Tribe

- A tech sub department containing a number of Chapters and Squads
- Has a tribe lead
- Every engineer belongs to one tribe

Chapter

- Group with similar engineering skills that share knowledge, tools and code
- Has a chapter lead, which is the closest manager to everyone in the chapter

Squad

- Work is done in a Squad
- Cross functional, autonomous teams, sit together and build something end-to-end
- Work is prioritized by the PO
- Connected to a specific project or long term mission in a product area

Guild

- A community of interest consisting of all associated chapters and everyone else who wants to join in (e.g. a QA guild)

Practical tips to enable agile methodologies in a global setup

- Add extra collaboration forums
- Keep teams in the same location
- Make sure the teams know about each other and share knowledge between teams
- Make sure to plan for close daily cooperation business people/developers
- Slice the elephant in a way that avoids dependencies to enable team autonomy

5. Exercises to Practically Experience Global Setup Challenges

6. Summary

- Global context adds challenges and opportunities based on **distance**, **time zones** and **culture**
- Utilize the positive effects of distance, time zones and culture
- Be aware of the **importance of communication**, consciousness in communication throughout the whole project
- Create teams and build your product in a way to enable autonomous teams – “slice the elephant”

- Add extra collaboration forums and enable informal communication
- Be stubborn

15 Agile Architecture

12 July - 18 July

Slides: Moodle

1. Motivation

- Product development and project management are different
- User Story Mapping is a good technique for providing initial rough estimations of a system
- Software architecture is importany
- Quality attributes should drive the architecture
- Software architecture can be managed continuously
- Duties of the *Software Architect* role

The world is changing fast ...

- Requirements may only be *hypotheses* and change quickly
- Current market trend is that *products* change to *services* (first mover often win)
- High uncertainty requires a different mindset than traditional environments (Lean Startup, Design Thinking)
- Agile software development and good software architecture enable the necessary flexibility

2. Product vs Project

In reality most IT projects are products. Work accordingly!

- IT Industry talks about "IT projects"
- However, in practice we work on systems as long as they are needed / valuable
  - most effort is long-term
  - things have to be good, flexible, etc. in the long term
- This requires **Product Management** not project management
  - Products are only successful with good Product Management
  - Focus on **value creation** not time & budget

3. Agile Release Planning

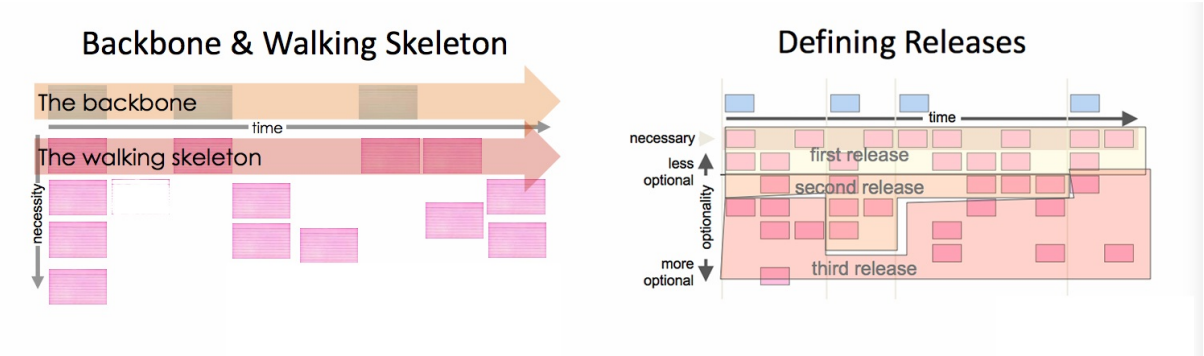
Personas

- Construct a profile containing information about the type of user relevant to the software being created
- choose specific characteristics and compiling those into a archetypal description
- based on assumptions

- most of the time you will have 3 - 5 personas

User Story Mapping

- Horizontal Axis: Time
- The Backbone
  - at top
  - Main Activities of Personas
- The Walking Skeleton
  - under activities
  - Tasks for each Activity / Persona
  - ordered by neccessity
- Create releases from coherent experiences



From Estimation Points to Time

- Have a closer look at 2 - 5 user stories
- Break them down as far as possible
- Guess the minimal / maximal person days for each task
- Assume that some uncertainty will be resolved by statistics: [min + 20% , max - 20%]
- **After 2-3 sprint reestimate the backlog based on experience**

Team velocity

- To get a better understanding of the team velocity check the velocity after 3-4 sprints, then
  - check your backlog for completeness
  - reestimate the backlog

Key Messages: Agile Release Planning

- Think in products, not projects
- User Story Mapping is a proven, fast & useful method to create a common understanding of a system
- All stake holders have after such a workshop a common vision for the product development start
- Together with the first few sprints this gives a solid base for planning

4. Agile Software Architecture

Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

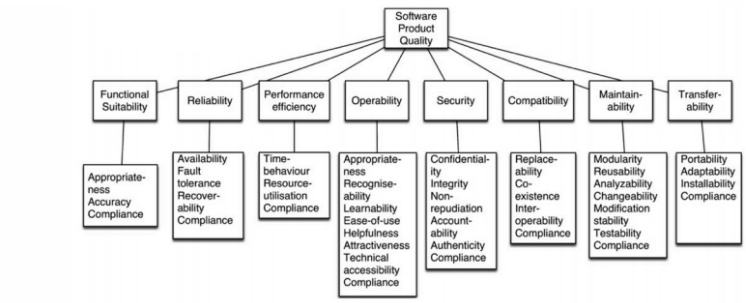
= those parts, which are they hardest to change

How Software Architecture can help

1. turns a big problem into smaller more manageable problems.
2. shows people how to work together effectively.
3. provides a vocabulary for talking about complex ideas.
4. looks beyond features and functionality.
5. connects business goals with the practical realities of implementation.
6. helps us avoid costly mistakes.
7. enables agility.

Architecture Drivers

A good architecture balances these (often conflicting) goals:



ISO 25010 Generic Software Quality Tree

Software Architect Role

- In small teams: architecture is often defined & evolved by senior developers
- In bigger product development team, the role is given to dedicated people
  - Architecture Agents
  - Supporting Architect
- Often intensive work together with UX and Business

Duties

- Clarify Requiremens
- Design Structures
- Design (crosscutting) concepts
- Communicate Architecture
- Shepard Implementation
- Evaluate Architecture

Methodically Designing Software Architectures

- iteratively design & decide
- ARC42: Architecture documentation

6 Principles of Continuous Architecture

1. Architect products, not just solutions for projects
2. Focus on Quality Attributes, not on functional requirements
3. Delay design decisions until they are absolutely necessary
4. Architect for change – „leverage the power of small“
5. Architect for build, test and deploy
6. Model the organization after the design of the system

What is good Architecture?

- Suits constraints that enable the long-term evolution of the software
- Solves business problems
- System remains easy to understand, change, extend
- Valuable

How to deal with large systems

Cognitive Mechanisms

- allow comprehension of larger system
- Chunking (Modularity)
  - responsibility, coupling, size, interfaces
- Hierarchies (Hierarchical Layering)
  - No cycles - neither on architectural nor on class level
- Schemata (Pattern Consistency)
  - Consistent and integrated pattern
  - Use a pattern language

Layering

- **Domain Layering:** Contracting, Offer/Ordering, etc.
- **Technical Layering:** Presentation, Business Logic, Database
- In the long run, domain layering is more important than technical layering

Timing of Software Architecture

- One Team: Continuously
- More Teams: Before start of development
  - Planned Design instead of Emergent Design

Current Trends

- Clean Code

- Design Patterns

Architectural Decision

Make good decisions:

– Create options, test them – Delay decisions as long as possible (until last responsible moment) – Try to make reversible decisions (if not too expensive) – Communicate them well!

Strategic Decisions

- Decisions spanning several subsystems or components
- Decisions involving compromises or tradeoffs
- Decisions involving intense stakeholder discussions
- Decisions founded in organizational or political constraints

5. Current trends: DDD and Microservices

Domain Driven Design

The critical complexity of most software projects is in understanding the domain itself.

- creates a common vocabulary
- bounded context: decoupling!

Architecture driven by DDD

- Clean Architecture
- (really good read, 10/10 can recommend)
- Article 8thlight

Microservices

- Monoliths put all functionality in on process and scales by replicating the monolith on several servers
- A microservice architecture puts each functionality into a separate service and scales by replicating those as needed
- Microservices are more complex to manage for **small projects**
- As complexity increases, productivity of monolithic architectures falls rapidly
- The decreased coupling of microservices reduces the loss of productivity for complex and large systems

6. Summary

Without Good Software Architecture

- Adding features will get more and more expensive
- Developers don‘t want to work on the system

- If you change one part of the system, a completly mentally unrelated other part of the system might get broken
- You can‘t use the system in other environments than the current one
- you have to think about rewriting the whole system (expensive, big risk) or get out of business..

6 Principles of Continuous Architecture

1. Architect products, not just solutions for projects
2. Focus on Quality Attributes, not on functional requirements
3. Delay design decisions until they are absolutely necessary
4. Architect for change – „leverage the power of small“
5. Architect for build, test and deploy
6. Model the organization after the design of the system
7. Think in products
8. Agile estimation and planning of products is possible, e.g. with User Story Mapping
9. Software Architecture is important to create and maintain the foundation of products and have options for the future

16 Release Management

12 July - 18 July

Slides: Moodle

1. Overview: Continuous Delivery and Release Management

1.1 Motivation

How it should NOT be done: Delivery only after the implementation was finished -> rare releases during development -> little feedback from clients / users during development -> "Throw a release over the fence" (dev environment != target environment)

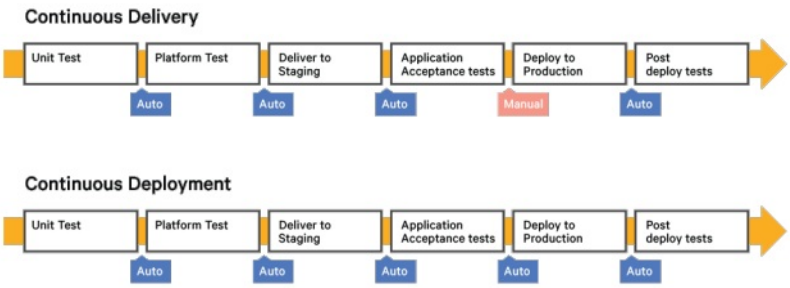
Release Management

- All activities concerning the creation (build) and the delivery or deployment (release) of software
  - compile source code and link with all dependencies to executable
  - package everything into an executable application
- **Goal:** Automate as many steps as possible
- Release management defines workflows, activities, best practices, how and when to create releases
- Today: mostly continuous integration and continuous delivery

1.2 Terminology

- **Continuous integration**: technique where members of a team integrate their work frequently. Usually each person integrates at least daily, leading to multiple integrations per day.
- **Continuous delivery**: approach in which teams keep producing valuable software in short cycles and ensure that the software can be reliably released at any time.
- **Continuous deployment**: every change that passes automated tests is deployed automatically.
- **Continuous software engineering**: organizational capability to develop, release and learn from software in short cycles.

Continuous Delivery vs Continuous Deployment



Continuous Delivery: Benefits and Challenges

Benefits

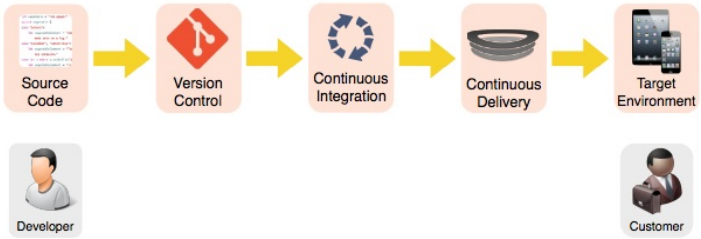
- Accelerated time to market
- Building the right product: improved product quality and customer satisfaction
- Improved productivity and efficiency
- Reduced risk of a release failure

Challenges

- Organizational: Varying interests in different departments
- Process: Traditional processes hinder continuous delivery
- Technical: Maintainability of the source code & tailorable delivery workflows for heterogeneous project environments

1.3 Release Management

The creation and delivery of releases to customers and users.



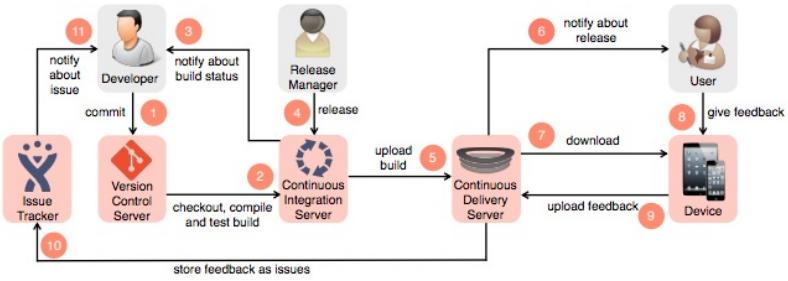
Release Manager

- Release Management is the process of managing software releases from development (changes to source code) to the user
- Responsibilities:
  - **Facilitator:** liaison between different business units to promote smooth and timely delivery of software products and updates
  - **Coordinator:** coordinate disparate source trees, projects, teams and components
- Development and Operations department need to work together => DevOps

DevOps

- Development + Operations
- Emphasizes on communication, collaboration, integration and automation
- The Release Manager acts as a liaison between the involved departments within the software development team

Example of a build and release management workflow with feedback



Not shown in this workflow: build failures, test failures, release failures

1.4 Build Management

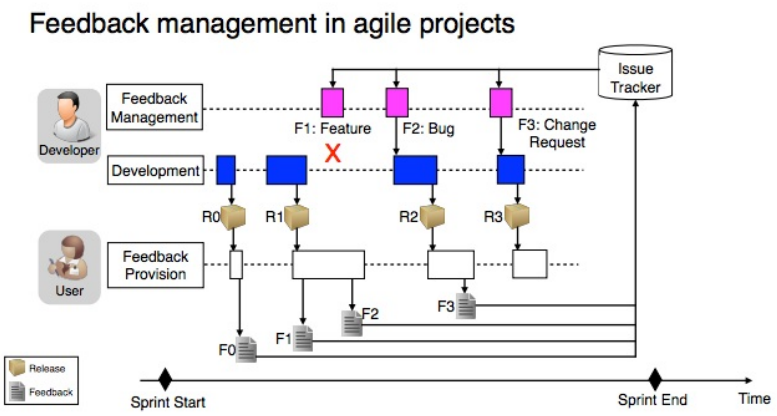
Build Management (continuous integration server)

- Regularly checks for changes published to the version control server (pull) or is notified when changes are available (push
- Builds the software
  - Compiles the source code
  - Executes the test cases (regression testing)
  - Packages the software
  - Code signs the software
- Notifies developers about build status
- Automatically detects multiple branches and creates a build plan for each branch (Bamboo)

Release Management (continuous delivery server)

- Distributes software to customers and users (a.k.a. Enterprise AppStore)
- Obtains user feedback about the usage of the software
- Collects crash reports
- Stores feedback and crash reports in issue tracker (Jira)

Feedback Management in Agile Projects



Release Notes

- Describe the most important changes to the product since the last release
- **Important:** Write them from the viewpoint of the user
  - no technical terms the user does not understand
  - no complex details about the system
  - high level content (short and readable)
- Delivered and Distributed when an update is released
- Shared with end users, customers and clients

2. Management Issues and Best Practices

Continuous Delivery

- How often do you deliver your application?
- Which process changes do you have to implement to make customers aware of multiple releases during development
- How do you handle customers who do not react to releases?
- How many releases can a customer handle?
- How do you handle feedback within the sprint?
- What needs to be done to implement continuous delivery from scratch? (CI -> CD, CI -> CD -> Feedback)
- How can you tailor the workflow to specific project needs?

3. Continuous Software Engineering

Best Practices for Build and Release Management

- The process for releasing/deploying software must be **repeatable** and **reliable**
- If something is difficult or painful, do it more often
- Automate **every step** of the build process
  - Write test cases
- Integrate your changes early and often
- Release early and often to obtain feedback

- Release internally first
  - Test it thoroughly
  - Then promote the release to your users
- **React to crashes and user feedback**
- Improve continuously

Why is Continuous Delivery difficult for large systems?

- Large systems tend to have many dependencies, legacy code, ...
  - hard to maintain and extend
  - integration testing takes a lot of time
  - each change can lead to many errors (due to dependencies)
- Loose Coupling and High Cohesion are hard to maintain over time
- The middleware layer become a **monolith** => system too big to maintain

Conway's Law

”Any organization that designs a system (defined broadly) will inevitably produce a design whose structure is a copy of the organization's communication structure.”

Typical Structure for Large Systems

- Monolithic Layered Architecture
  - Organization in specialized teams: UI, Middleware, DBAs
  - Many different services in middleware layer
- Better Approach: Microservices
  - Organization: cross functional teams
  - Each microservice only contains 1 service with functionality
- **Benefits:**
  - Strong Module Boundaries (modular structure)
  - Independent Deployment (simple services are easier to deploy)
  - Technology diversity (possibility to mix languages / development frameworks)
- **Costs:**
  - Distribution (harder to program, remote calls have risk for failure)
  - Eventual Consistency (maintaining strong consistency almost impossible)
  - Operational Complexity (mature operations team needed)

4. Summary

- **Build and Release Management** are important topics for today's software engineering projects
- Managers need the ability to use the concepts and tailor them to specific projects environments
- **Continuous Integration**
  - Continuously **build** (compile, integrate, test) the source code base
  - Continuously test changes to find regressions
- **Continuous Delivery**
  - Continuously **release** (i.e. deliver) applications to users

- Continuously beta test applications with customers / users
- Quickly obtain feedback to realize short development feedback cycles

17 Risk and Demo Management

19 July - 25 July

Slides: Moodle

1. Risk Management

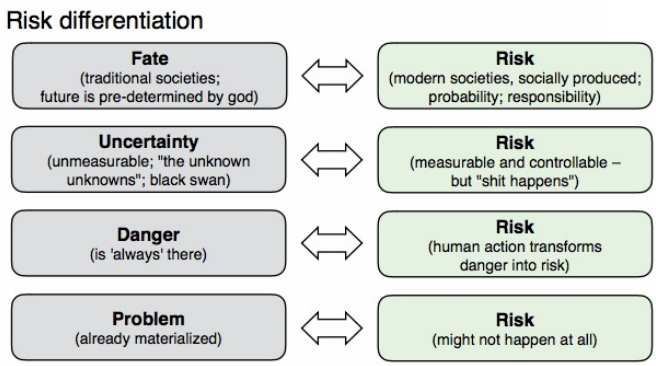
Risks must be taken because the greatest hazard in life is to risk nothing.

Risk

- ... is the deviation of a result of a future event from expectations
- ... is uncertain whether it will materialize or
- ... is a potential harm that may arise of such an event
- ... depends on our decisions

Murphys's Law: *"Anything that can go wrong, will go wrong"*

Risk Differentiation



Risk Assessment

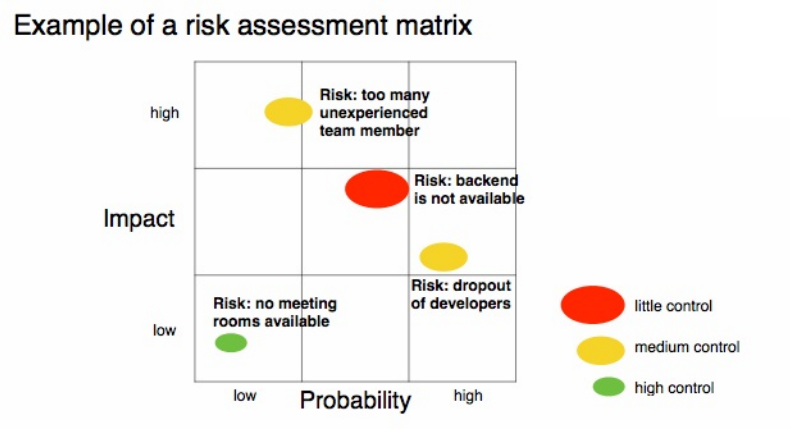
Important: Accept the uncertain, but identify risks!

Identify Risks

- What might affect the project objectives? (internal / external)
- Performed by project management AND team
- **Methods:** brainstorming, checklists, expert discussions, SWOT analysis
- Categorize risks (complexity, technology, communication, ...)
- Assign risk triggers
- In agile Projects (Scrum)
  - Daily Scrum: status, **impediments**, promises
  - Sprint Review: identify improvements for next sprint

Risk Assessment Matrix

- 3 Dimensions: Impact, Probability, Control



Risk Mitigation

Risk Mitigations can be grouped into 4 possible types.

1. **Avoidance**
   - Try to avoid risk occurrence
   - Mostly used at planning phase
2. **Control**
   - Identify actions to control risk occurrence and impact
   - Most common response
3. **Transfer**
   - Delegate risk (e.g. to subcontractors)
4. **Investigation**
   - Try to analyze risk further
   - Risk is not reduced

Risk Mitigation Plan

- Periodic Risk Assessment
- Possible Escalation to upper management

Risk Management Skills

1. **Rules**
   - Update risk log
   - Establish contingency plan
2. **Knowledge**
   - Specific IT knowledge
   - Business / functional know how
3. **Distance**
   - Challenge time and cost estimates
   - Listen to different opinions

4. **Intuition**
    - Expert Estimations
    - Feeling about risks and problems

2. Demo Management

Motivation

- Boring demonstrations do not convince your customer for future investments or users to buy your software
- Decisions are heavily influenced by feelings
- Combination of technical and marketing skill required

How to make a great, convincing and entertaining demo?

- Selection of the right demo: focus on the core value to end users
- Influenced by the real world usage of the software
- Described by visionary scenarios in the problem statement

2.1. Tornado Model: From Visionary Scenario to Demo Scenario

Scenario-Based Design

Start with the problem statement given by the customer.

- Section 2 (Scenarios) describes the visionary scenario formulated by the customer
- Developers transform the visionary scenario into formalized scenarios The formalized scenario is used as a basis for the demo

Formalized Scenarios

Formalized scenarios consist of 6 components

1. **Scenario Name:**
    - Noun + Verb
2. **Participating Actors:**
    - concrete users of the system including a role
    - e.g. "Bob: Employee"
3. **Flow of Events**
    - describes interaction between the actors and the system
    - indentation helps to distinguish user input from system response
4. **Entry Conditions**
    - have to be met before the flow of events can start
5. **Exit Conditions**
    - need to be true at the end of the flow of events
6. **Quality Requirements**
    - non-functional requirements of the scenario

Write a screenplay from the formalized scenario(s)

Identify participating Methods / Objects

- Inspect the flow of events in the screenplay
- Textual Analysis (Abott's Technique)
- Create a class diagram from the flow of events
- Other knowledge sources
  - Application knowledge (application domain, taxonomies)
  - Solution knowledge (e.g. design patterns)
  - General world knowledge

2.2 Mock Object Pattern

Can we do a demo with a missing subsystem?

Yes! We use **model based demonstration:**

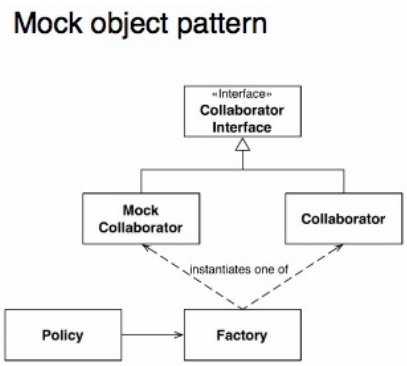
Model-Based Demonstration

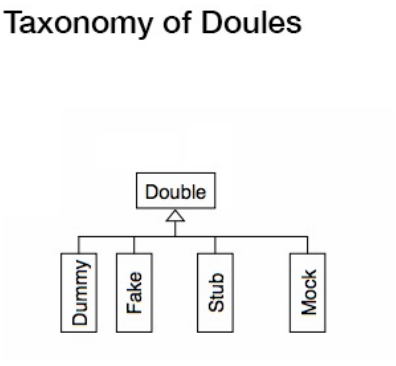
1. We start with the **System Model**
2. We identify the **SUD** (System under Demo)
3. Find other objects that interact with the SUD, but are not implemented yet. These are **Collaborators**.
4. To interact with Collaborators, create **Mocks**
5. **Mocks** mimic the behavior of the Collaborators

Mock-Object Pattern

Good design is crucial when using mock objects: the real object (subsystem) must be specified with an interface (façade) and a class for the implementation.

- The mock collaborator replaces the behavior of the collaborator (the real object)
- A mock object can be created at startuptime with a factory pattern
- Mock objects can be used for testing state of individual objects as well as the interaction between objects





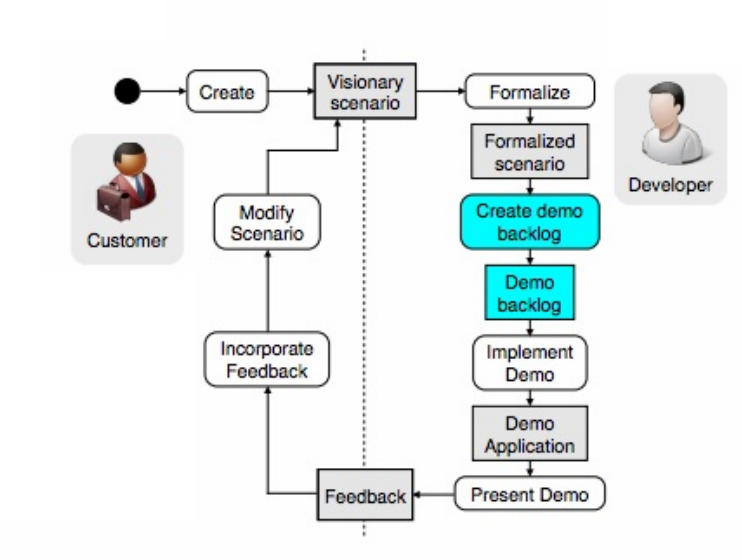
Taxonomy of Doubles

- **Dummy object:** often used to fill parameter lists, passed around but never actually used
- **Fake object:** a working implementation that contains a “shortcut” which makes it not suitable for production code
- **Stub:** Provides canned answers to calls made during the test. Provides always the same answer
- **Mock object:** mimic the behavior of the real object. Know how to deal with a specific sequence of calls they are expected to receive

Software Demo Tips

- Be enthusiastic and convincing, but don’t lie
- Have a strong opening
- Include humor (ideally subtle but funny jokes), but don’t make fun of somebody
- Focus on user needs and problems that the software solves
  - Find the right balance between marketing and technical complexity
  - Explain technical decisions and details in a way that even non IT people can understand it
- Prefer application domain terminology instead of solution domain terminology
- Prepare backups (videos, pictures) in case something goes wrong (Murphy’s law)
- If something goes wrong, accept the failure and deal with the situation

2.3 Demo Feedback



- Developers deliver the demo application to the customer
- They present the demo application and request feedback from the customer
- The feedback is then used to refine the initial visionary scenario (or design)
- Notice: developers should be honest about what is implemented and what is mocked

3. Summary

- Preparing a great demo of your application convinces the customer

    - Play the role of the user of the application in an authentic way, e.g. dress up like the real user
    - Use humor in your demonstrations (but: don't overdo it)
- Prepare a screenplay with the scenes of the demo (based on the event-flow of the demo scenario)
- Use the mock object pattern to mimic the behavior of collaborating objects that are not implemented yet
- With a great demo, all stakeholders understand the purpose of the application
- Remember Murphy's Law and prepare for it: *"anything that can go wrong, will go wrong"*
    - Use risk management and have a backup plan if risks materialize as problems
    - Prepare backups for your demo, e.g. videos and images

18 Project Management Antipatterns

26 July - 1 August

Slides: Moodle

1. Pattern vs Antipattern

Definition: Pattern

To be considered a pattern, a solution must be applicable to more than one specific problem.

1. **Problem:** eloborates
    i. one **context**
    ii. a set of **forces**
2. **Solution:** resolves these forces with
    i. several **benefits**
    ii. several **consequences**
    iii. solutions can generate **follow-on-problems**

Important: Patterns can become antipatterns (problematic solutions), when change occurs.

- Change of requirements
- Change of project parameters
- Change of methodology

Definition: Antipattern

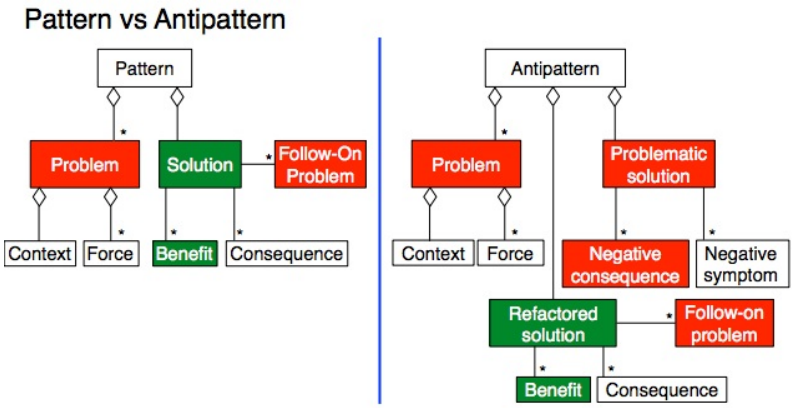
Antipatterns identify and categorize common mistakes in software practice. Informal synonyms are

- Bad Practice
- Bad Idea or Pitfall
- Typical PM mistake
- Typical design error
- Bade use of an acitivity

Antipatterns consists of **1 problem** and **2 solutions**

1. **Solution**
    i. one **context**
    ii. a set of **forces**
2. **Problematic Solution** describes a commonly occuring solution that causes
    i. multiple **negative consequences**
    ii. multiple **negative symptoms**
3. **Refactored Solution** describes how the problematic solution can be reengineered and lead to

i. multiple **benefits**
ii. multiple **consequences**
iii. potential **follow-on-problems**



Applicability of Patterns and Antipatterns

- **Patterns** are good for problems, which do not have a solution yet
- **Patterns** emphasize the use of proven good design principles
- **Antipatterns** are good for emphasizing the recognition of mistakes

Changing bad solutions into better ones

- When a pattern becomes an antipattern, it is useful to have a proven approach how to change the problematic solution into a better one.
- **Incremental Reengineering** (refactoring)
  - process of incrementally changing the structure of a bad system / project / organisation into a better one with the use of antipatterns

Why antipatterns?

- The presence of good patterns alone in a successful system is not enough
- You must also show that those patterns are absent in failing systems
- Likewise it is useful to show the presence of certain patterns in failing systems and their absence in successful systems

2. Typical Mistakes in Software Engineering

How to kill a project!

- Show the same demo twice to the same audience
- Focus on technologies, not on problems and scenarios
- Don’t maintain consistency between releases
- Isolate team efforts from other teams within an organization
- Rewrite existing clients, servers and applications
- Change the purpose of the system so that the models describe the wrong objects

Root Causes for Antipatterns

Most common mistakes in software project management and development include:

- Insufficient communication with the client
- Unfulfilled requirements
- Insufficient testing
- Cost overruns and schedule slips

Seven Deadly Sins of Software Development

1. Haste

- Solutions based on hasty decisions (“time is most important”) lead to compromises in software quality

2. Pride (hubris)

*Not invented here (NIH): Not willing to adopt anything from the outside

3. Apathy

*Not caring about a problem, followed by unwillingness to attempt a solution

4. Sloth

- Making poor decisions based on “easy” answers

5. Ignorance

- Failure to seek understanding

6. Avarice (excessive complexity)

- No use of abstractions, excessive modeling of details

7. Narrow-mindedness

- The refusal to use solutions that are widely known (“Why reuse? I only have to solve one problem”)

3. Taxonomy of Antipatterns

Roles: Developer, Architect, Manager

Development Antipatterns

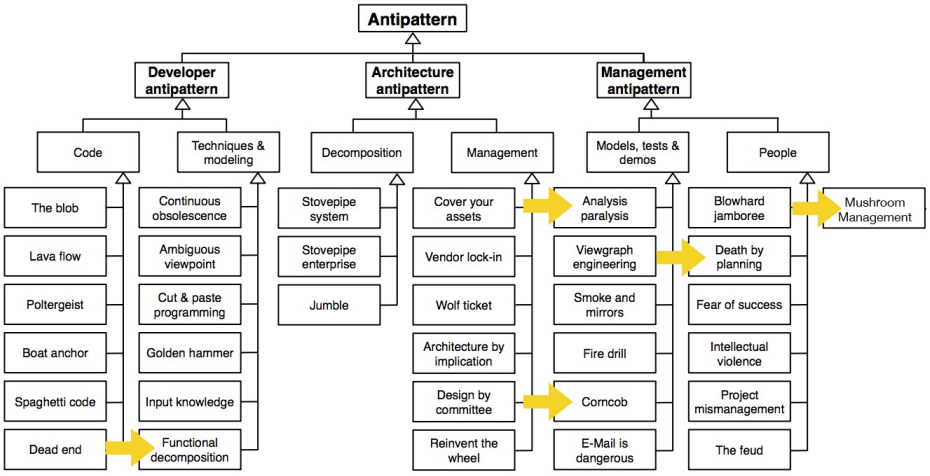
- Perspective: Software Developer
- Issues: Software refactoring, modification of the source code to improve structure with respect to the long term maintainability

Architecture Antipatterns

- Perspective: Software Architect
- Issues: Partitioning of subsystems and components, platform independent definition of interfaces and connectivity of components

Management Antipatterns

- Perspective: Software Project Manager
- Issues: Software Project Organisation, software process model, human communication, rationale management and resolution of issues



4. Examples of Antipatterns

4.1 Analysis paralysis

Unrealizable models are worth nothing.

General Form

- Goal to achieve perfection and completeness of the analysis phase
- Generation of very detailed models
- Detailed analysis can be successfully completed prior to coding
- Assumption, that everything about a problem can be known a priori
- Analysis model will not be extended nor revisited during development

Symptoms and Consequences

- Cost of analysis exceeds expectation without a predictable end point
- Analysis documents no longer make sense to the domain experts

Typical Causes

- Management assumes a waterfall progression of phases

Refactored Solution

- Iterative development
- Incremental development assumes that details of the problem and its solution will be learned in the course of the development process
- Vertical prototyping
- Scenario based design
- Sprint based development

4.2 Functional Decomposition (No OO)

General Form

- everything is a function, lots of files called misc, util, util1, aux1, aux2…
- **Unbalanced Forces:** management of complexity, change management is difficult

Symptoms and Consequences

- The functionality is spread all over the system
- Maintainer must understand the whole system to make a single change
- Code is complex, hard to understand, high coupling between code sections in different files, impossible to maintain
- User interface is often awkward and non-intuitive

Typical Causes

- Programmers have been trained only in an imperative / functional language
- Designers have been trained with a functional decomposition method

Refactored Solution

- Object-oriented analysis
- Object-oriented reengineering (process)

4.3 Corncob

Also known as: Corporate shark, loose cannon, TWIT (Third World Information System Troubles)

General Form

- A difficult person (the corncob) causes problems for a software development team due to destructive behavior
- Corncobs focus much more on politics than technology
- They are usually experts at manipulating politics

7 Types of difficult people

1. Hostile Aggressives

   i. Sherman Tanks
   ii. Snipers
   iii. Exploders
2. Indecisives
3. Whiners
4. Negativists
5. Clams
6. Bulldozers
7. Superagreeables

Symptoms and Consequences

- A project is unable to make progress because someone disagrees with key objectives or essential processes and continually tries to change them
- Political forces create an environment where it's difficult to keep technical discussions on track
- Political forces result in frequent changes to the scope or requirements of the system

Typical Causes

- Pride, avarice, ignorance, narrow-mindedness
- The corncob has a **hidden agenda**, which conflicts with the team's goals
- There is a fundamental disagreement between the corncob and the team members
- Management has inappropriately allocated roles to staff, who abuse them for theirown ends

Known Expections

- Is acceptable when a company or product development manager is willing to live with the actions of the corncob
- In projects that involve multiple organizations, it's sometimes useful to have a designated corncob, whose role is to defend an existing architecture from inappropriate changes

Refactored Solution

- **Transfer the responsibility:** turn the responsibility for planning and resolution over to the person who raises the concerns
- **Question the question:** when the corncob uses ambiguous or "loaded" words or phrases, ask him or her to clarify their meaning
- **Corrective interview:** management meets individually with the person causing the problems, and explains the impact of his or her behavior
- **Friendly outplacement:** recommend a corncob to employment headhunters to help him or her to exit gracefully
- **Corncob support group:** if there are several corncobs in an organization, management can transfer them into the same group

4.4 Death by Planning

General Form

- many projects fail from either over planning or under planning
- **Glass case plan:** gives the management a ‘comfortable view’ of delivery, often before the project starts.
  - Management assumes everything will happen according to plan
  - Progress is neither checked nor tracked.
- **Detailitis plan:** over planning in projects, resulting in delays, staff attrition, and project failure
  - Effective Delivery is assumed to be achieved by a high degree of control via continuous planning
  - Frequently evolves in a hierarchical sequence of plans with unnecessary level of detail
  - Updating the plan takes time away from those who should implement it

Symptoms and Consequences

Glass case plan

- **Primary symptom:** lack of an up-to-date project plan
- **Secondary symptom:** no up-to-date plan —> increase in differences between management expectations and reality (extension of the primary symptom)
- **Consequences** grow incrementally, with the options of:
  - Lack of status quo of the project status
  - Cost overrun
  - Attrition
  - Failure to deliver a critical deliverable (the final consequence)

Detailitis plan

- **Symptoms**
  - A lot of time is spent on planning, capturing progress, and replanning
  - Often more time than on delivering the software
- **Consequences**
  - Endless planning and replanning causes further planning and replanning
  - Objective shifts from delivering software to delivering plans
  - Delays in software delivery lead to the eventual project failure

Typical Causes

- **Root causes:** avarice, ignorance, haste
- **Causes** for the glass case plan
  - Overambitious initial planning to attempt to enforce absolute control of project
  - Ignorance of basic project-management principles
  - Lack of a pragmatic approach to planning, scheduling, and capture of progress
- **Causes** for the detailitis plan
  - Overambitious continuous planning to attempt to enforce absolute control of the project (development)

    - Planning as the primary project objective
    - Ignorance of change

Refactored Solution

- **Known exceptions** Software development projects can succeed despite poor planning practices
    - The existence of a “project champion” within the management level can prove beneficial even in the context of bad planning
- **Refactored solution**
    - Create schedule of related development tasks defined by a software development lifecycle model, e.g. use Gantt charts to visualize deliverables, dates and interdependencies
    - Estimate the tasks in terms of effort and elapsed time
    - Capture the progress of tasks for an elapsed time period (use burn down chart)
    - Evaluate captured progress against the project plan
    - Update the captured progress against the project plan

4.5 Mushroom Management

Also known as: Pseudo-Analysis, Blind Development

General Form

- Requirements change frequently

Symptoms and Consequences

- **Unbalanced Forces:**
    - Management policy isolates the developers from the system’s end users
    - Management assumes that the requirements are stable and well understood by developers

Typical Causes

- Bad understanding of the requirements leads to poor design decisions
    - Developers must make assumptions, which may lead to pseudo-analysis
    - Pseudo-analysis: takes place without end-user participation
- Communication channels do not work properly (between managers and employees)

Refactored Solution

- Introduce **incremental** and **iterative** development
    - Incremental and/or iterative development based upon prototyping and user feedback
    - Every project increment includes extensions to the user interface functionality
    - Incremental development reduces the risk of the project not being accepted
- Include a domain expert in the development team
    - **Follow on Problem:** the domain expert represents only one opinion of the domain community
- Include a user in the development process

- **Follow on Problem:** in innovative projects the end user may not exist

5. Summary

Example Antipatterns

- **Analysis paralysis**
  - Spending excessive time in requirements elicitation and analysis
- **Functional decomposition**
  - Design and implementation style of developers, who got their initial training in structured analysis methods and imperative languages, with no or little experience in OO methods and OO languages
- **Corncob**
  - Difficult people who usually create additional stress
- **Death by planning**
  - Too much planning, lack of an up-to-date project plan
- **Mushroom**
  - Bad understanding of requirements, management isolates developers from the system's end users

POM - Question Catalogue

TUM #POM #tbd

1. Quiz Questions

Quiz 1: Introduction

Question: How did the Columbus egg problem end?
a. Columbus destroyed the egg
b. The egg stood on its end
c. The problem is not yet solved
d. The other people found a solution to the problem
Question: Which goals of a software project define its success in modern project management?
a. Extend the deadline
b. Develop High Quality
c. Adapt to changes
d. Exceed the budget

Quiz 2: Team Building

Question: The roles Scrum Master, Product Owner and Developer describe what kind of project?
a. Agile Project
b. Heavyweight Project
Question: Which of the following stages are used to describe team development?
a. Hunting
b. Storming
c. Fighting
d. Forming
Question: Let's recap the differences between groups and teams.
a. A [Team] has a common goal.
b. Participants of a [Group] are loosely connected.
c. Each [Team] is a [Group], but not each [Group] is also a [Team].
d. [Team] members need to work together.

Quiz 3: Project Organization

Question: What is the defining characteristic of a project function in comparison to other activities?
a. Project functions contain other low-level activities
b. A project function spans the entire duration of the project

c. Project functions are activities that require an experienced developer
d. A project function culminates in a milestone
Question: What are main responsibilities of the project manager?
a. Maintaining accountability
b. Analyze the application domain
c. Design and implement tests
d. Administering the resources
Question: Which of the following is NOT a unit of work?
a. A work package
b. A deliverable
c. A task
d. An activity
Question: What are typical challenges in a matrix organization?
a. Conflicting interests between department and project
b. Projects cannot be tailored around the problem
c. Competing demands for team members in multiple projects
d. Teams cannot be assembled rapidly
Question: What are typical organization forms?
a. Delegation Organization
b. Matrix Organization
c. Line Organization
d. Project-based Organization
Question: What are characteristics of information flow in non-hierarchical project organizations?
a. It is easier to manage because of the simplified communication structure
b. It increases bureaucracy due to the complicated decision flow
c. It expects decisions to be made at each level
d. It reduces development time because the communication is more efficient
Question: What are typical roles in meeting management?
a. Primary Facilitator
b. Scrum Master
c. Minute Taker
d. Product
Question: What are characteristics of European and Polynesian Navigation?
a. Agile Planning as in the Polynesian Navigation uses context dependent actions
b. Traditional Planning as in European Navigation corrects the course to follow the initial plan when a deviation is observed
c. Traditional Planning as in the European Navigation can easily react to changes
Question: Which of the following is an example for an asynchronous communication mechanism?
a. Stand-up Meeting
b. Slack
c. Informal Meeting
d. Confluence

Quiz 4: Software Lifecycle Models 1

Question: Which managerial challenges come with modeling?
a. Complexity
b. Improved communication
c. Redundancy
d. The Analysis Paralysis Antipattern
Question: What are possible changes when tailoring a software lifecycle model to specific needs?
a. Leave out the activity "Maintenance"
b. Leave out all activities
c. "Test" before "Implementing" the system
d. Rename "Detailed Design" to "Object Design"
Question: Which of the following statements about modeling are correct?
a. Conceptual models represent a person's perception of an idea
b. Specification models communicate the idea to a tool
c. A mockup drawn on a napkin is a specification model
d. The best way to support communication is formal modeling
Question: Which of the following statements about software lifecycle models are correct?
a. XP, Kanban, Scrum and Unified Process are agile models Incorrect
b. Extrem Programming is based on feedback loops
c. V-Model and V-Model XT are both sequential models
d. The Waterfall Model is the current state-of-the-art practice
Question: Which of the following software lifecycle models are iterative models?
a. Spiral Model
b. V-Model
c. Waterfall
d. Unified Process
Question: Which process steps are part of the Waterfall Model?
a. Design Correct
b. System Testing
c. Installation
d. Unit Testing

Quiz 5: Scrum

Question: Which statements about Scrum teams and roles are correct?
a. The development team works on items in the Sprint Backlog
b. The Product Owner is responsible for the process
c. The Scrum Master prioritizes the backlog items
d. The development team is self-organizing
Question: Which statements about Scrum teams and roles are correct?
a. As a student I want to access my course schedule so that I am on time in class
b. As a kid I want to have a slide so that I can use it.
c. As a cook I want a system that is connected to my fridge.

d. I want a possibility to determine the scale of discount of a customer.
Question: Which of the following are Scrum artifacts?
a. Potentially Shippable Product Increment
b. Daily Scrum Meeting
c. Product Kickoff Meeting
d. Product Backlog

Quiz 6: Software Lifecycle Models 2

Question: What is correct about iterative and incremental development?
a. Incremental development only changes existing functionality
b. Refactorings can cause product iterations
c. Iterative development aims to improve existing system parts
Question: What is correct about prototypes?
a. Prototypes are used in the Waterfall model
b. Vertical prototypes only cover one layer of the system
c. Rapid prototyping is a misnomer
d. Prototypes are used in the Spiral model
Question: What is correct about verification and validation?
a. Both techniques are internal processes
b. Verification evaluates whether you are building the system in the right way
c. Both techniques are used in the V-Model
d. Validation assures that you are building the right system
Question: Which of the following workflows are supporting workflows in the Unified Process?
a. Configuration and Change Management
b. Analysis and Design
c. Deployment
d. Project Management
Question: Which of the following statements about agile software lifecycle models are correct?
a. It is difficult to introduce agile methods in a company that is only used to sequential lifecycle models

b. Agile methods are not recommended for small projects with frequent changes
c. In Test Driven Development (TDD), developer first implement a functionality, then refactor it, and finally test it
d. The deliverable in Extreme Programming (XP) is comparable to a Product Increment in Scrum
Question: Which practices are used in Extreme Programming (XP)?
a. Write production code with two people
b. Improve understandability of source code while leaving its behavior unchanged
c. Divide a project into the engineering and production stage
d. Write a test before implementing a feature, then refactor

Quiz 7: Usability Management 1

Question: Which terms are correlated to Usability Management?
a. Error handling
b. Portability
c. Memorability
d. Learnability
Question: Which statements about user interfaces are correct?
a. The effort for the development of good user interfaces is high
b. The more controls a user interface includes, the better is its learnability
c. User interface development requires interaction with users
d. All error messages in the user interface should be intended for the developers
Question: Which term does the following sentence describe? ”A measurement of how easily a user can use an application"
a. Usability
b. Accessibility
c. Functionality
Question: What are characteristics of a good heuristic evaluation?
a. All problems are justified properly
b. In general, there is no need to consider any other heuristics other than Nielsen's 10
c. The interface is evaluated multiple times
Question: Which heuristics need to be considered regarding consistency and standards?
a. Similar buttons should do similar actions
b. All options must always be visible to the user
c. Actions should be performed in a similar way as the target platform's actions
Question: Which heuristics have to be taken into consideration when reporting errors to users?
a. Show a detailed report with all technical details of what went wrong
b. Offer constructive help with the error and explain why it occurred
c. Be as precise as possible and include the user input
d. Blame the user

Quiz 8: Usability Management 2

Question: Which statements about human capabilities in UI development are correct?
a. Long term human memory has a large capacity and little decay
b. As a consequence of Fitt's law, it is recommended to use hierarchical menus in UI design
c. Fitts's law describes how fast a user can move his hand to a target of a certain size at a certain distance
Question: What are challenges in prototyping?
a. Prototyping does not allow iterations
b. A prototype with many problems can lead to negative feelings of the software
c. Prototyping always increases the costs Incorrect
d. Customers might have wrong expectations after they have seen a prototype

Quiz 9: Software Configuration Management 1

Question: Promotion vs. Release vs. Revision. What is correct?
a. A potentially shippable product increment is a promotion
b. A release is a version made available to users or clients
c. Revisions change major functionality of the developed system
d. Promotions are distributed only team internally
Question: What is correct about the object model for configuration management?
a. Controlled items are modeled with the composite pattern
b. The master directory contains all the promotions
c. A configuration item can have multiple versions
d. A configuration item can be a version
Question: Which of the following are Software Configuration Management activities?
a. Promotion Management
b. Risk Management
c. Branch management
d. Configuration item identification

Question: What are bad practices in distributed version control?
a. Keep the working copy up to date
b. Commit changes for two different bugs at once
c. Promote changes often to avoid merge conflicts
d. Change promoted versions in the remote repository
Question: What is correct about Git commands?
a. To promote changes you use **git commit**
b. **git pull** is a compound command composed of fetch & merge
c. **git clone** creates a local working copy of an existing git repository
d. **git add** adds file changes in your existing directory to your index (staging area)
Question: What are bad practices in distributed version control?
a. Administrators have total control over who can do what
b. The master directory can be restored from a programmer's directory if the server crashes
c. Branches are lightweight
d. It has a low learning curve
Question: What are characteristics of change requests?
a. Each change request includes a prototypical implementation of the solution to support the assessment of the request
b. Change requests are only approved if they are easy to implement
c. Each change request triggers a change request process
Question: What is NOT a typical activity in a change control process?
a. Create a new baseline
b. Evaluate the change request
c. Ask a developer for permission
d. Implement the change

Quiz 10: Developing Winning Proposals

Question: What is true about the content and effort of proposal types?
a. Co-sourcing may include resource options in a controlled manner
b. Outsourcing may include resource options in a controlled manner
c. Venture has defined deliverables and outcomes
d. Consulting has an indefinite duration
Question: What is correct about proposals?
a. The proposal is part of the final contract
b. In real life, IT proposals are mostly limited to design and implementation projects
c. A proposal is about a specific problem, but typically does not include a description of the solution
d. A proposal does not include legal statements
Question: Which are key attributes of a proposal?
a. Flexible
b. Tailored
c. Formal
Question: What makes a winning proposal?
a. Focus the proposal on the procurement department
b. Offer a good price
c. Gain the trust of the client
d. Promise the client to deliver the best possible solution
Question: The following text is a part of an already existing proposal:
"...At the end of the project we are providing the best software running on a new, faster and fully functional server. In addition, all of the existing data, in an optimal way, have been moved from the old server to the new. All of the current functionality still exist. Access to the system...."
Is this proposal acceptable?
a. Yes
b. No
Question: What are good reasons to decline bidding for a proposal?
a. Previous similar proposals made to this client were all rejected
b. You currently do not have the required skills or capacity
c. You already have a lot of business with this client
d. A lot of interaction with the client is required to create the proposal

Quiz 11: Kanban

Question: A Kanban system is an example of a ...
a. Drum-Buffer-Rope system
b. Pull system
c. Push system
Question: Which statements about "Kaizen" and waste are correct?
a. Consistent and need-based capacity utilization is the goal when reducing waste from inconsistency and inhomogeneity.
b. Reducing wait time is an example for improvement through reduction of activities that don't add value.

c. Kaizen aims to revolutionize the old process by replacing it with a new one.
Question: What should the Kanban board reflect?
a. What the team actually does
b. The process of the team as it is specified in official project documents
c. It depends
d. The desirable process of the team
Question: What does Kanban focus on when introduced to teams with an existing development process?
a. Changing roles and responsibilities
b. Evolutionary change
c. Leadership
d. None of the mentioned
Question: In a Kanban project, the development team is blocked, because the deployment team has hit the deploy WIP limit. If they would start to work on an additional task, they would break the development WIP limit. What should the development team do according to Kanban?
a. Increase the WIP limit of the development team
b. Increase the WIP limit of the deployment team
c. Help the deployment team to solve the problem
d. Break the development WIP limit and start another task
Question: How does a WIP reduction affect the lead time?
a. Lengthens lead time
b. Shortens lead time
c. Has no effect

Quiz 12: Software Configuration Management 1

Question: Which statements about pull requests are correct?
a. When the quality of the changes in a pull request is not ok, the reviewer must implement the requested improvements
b. In the code review workflow, the reviewer approves the pull request and then merges the changes of the feature branch to the development branch.
c. When a developer creates a pull request, and then commits further changes to the source branch of the pull request, the pull request is automatically updated
Question: Which statements about pull requests are correct?
a. Poor code in the development branch instills a sense of abandonment and spreads out to the rest of the source code
b. It recommends that developers should use test driven development
c. It describes how to avoid the phenomenon of software entropy
d. Poor but working code should stay unrepaired, otherwise you loose too much time
Question: What are best practices for branch and merge management?
a. A good practice to avoid merge conflicts is to pull often from the development branch into the feature branch.

b. Avoid having feature branches that span through a long period of time.
c. Make use of a specific branching model.
d. All requirements should be mapped to one feature branch, regardless of their size.

Quiz 13: Contracting

Question: Select all acceptance-relevant aspects to be considered in contracts.
a. Validation of the project results
b. Verification of the project results
c. Check for completeness
Question: For contracting parties, it is important to know ...
a. Once the contract is concluded, contractors have to implement quality management procedures to meet all project goals defined in the iteration plan
b. Once the contract is concluded, clients just receive deliverables and confirm timely shipment.
c. There are always two roles (client and contractor) that have individual goals and approaches.
Question: Contracts, in general, define ...
a. The responsibilities of the respective contracting parties
b. The basic project parameters in terms of deliverables, schedule, and budget.
c. How quality requirements regarding real-time are implemented.
Question: Select an interaction strategy that should guide contracting procedures as well as project operation in general to establish a fair set up.
a. Competitive (my win is your loss)
b. Indifferent (I don't care - you lose)
c. Cooperative (win-win)
Question: For the different contract types, select the statements that apply:
a. Fixed-price contracts do not allow for defining any incentives
b. Fixed-price contracts are the option preferred by clients to minimize their project risk
c. Using a time & material contract, clients face the risk to never get a working system
Question: Regarding the practices to embody contracts, the following statements apply:
a. Money for nothing is an incentive model awarding good performance
b. Maximum price is a means to better control the project budget by setting limits
c. The agile fixed price combines many proven practices thus fully replacing the classic fixed price

Quiz 14: Estimation and Scheduling

Question: What should be considered when estimating?
a. Complexity of the project
b. Project duration
c. Project budget
d. Infrastructure
Question: What is correct about methods for estimation?
a. COCOMO is an expert estimation technique with a top-down approach
b. Planning poker is a team expert estimation technique
c. Expert estimations are based on formulas

d. The lines of code estimation is considered as malpractice

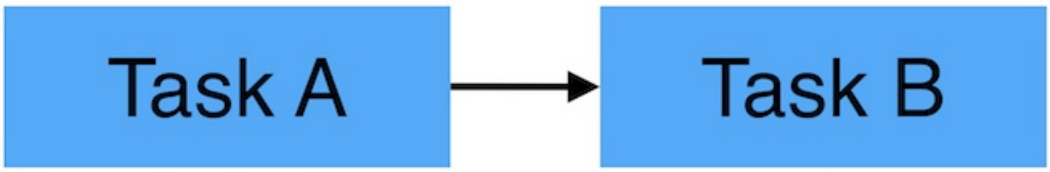
Question: What are challenges of planning poker?

a. Different teams estimate differently

b. It helps to discover important project issues

c. Estimates can vary based on the abilities of the people who are estimating

Question: Which type of dependency is shown in the following picture?



a. Finish-to-Finish

b. Start-to-Finish

c. Start-to-Start

d. Finish-to-Start

Question: What are good strategies to shorten the project duration?

a. Consider to use existing components

b. Find activities that can be parallelized on the critical path

c. Add manpower to the project

Question: What does a good project manager do?

a. Establish good and clear communication only with the best employees

b. Stimulate independent thinking

c. Avoid assumptions

Quiz 15: Build Management

Question: What is correct about continuous integration?

a. The build process usually includes multiple steps

b. High complexity of software systems makes it harder to integrate the software components

c. The later integration occurs, the less likely it is that unexpected failures occur

d. The continuous integration server notifies developer about the build status

Question: What are typical fault handling techniques?

a. Fault detection: testing

b. Fault ignorance: ignore all faults

c. Fault tolerance: software configuration management

d. Fault avoidence: software configuration management

Question: Choose the correct answers about testing.

a. Integration testing tests the interfaces between subsystems

b. System testing confirms that classes and methods are correct
c. During a beta test, software is tested as realistic as possible in the target environment
d. Acceptance tests are performed by the client, not the developer

Quiz 16: Global Project Management

Question: Estimate how many large IT-projects (those with initial price tags exceeding $15 million) go so badly that they can threaten the very existence of the company?

-> No correct answer given. I think it was 6%.

Question: Communication is crucial. How can it be leveraged to reduce challenges specific for global projects?
a. Hold isolated team meetings only
b. Usage of appropriate communication tools
c. Plan for both asynchronous and synchronous communication
d. Enforce formal and trackable communication
Question: Which of the following mitigations solve problems in global setups?
a. Improvement epics in the backlog
b. Additional time for innovation between sprints
c. The introduction of collaboration forums to improve the autonomy of teams in later phases of the product life cycle
d. Teams should present their delivery for other teams to prevent isolated teams
Question: What are key success factors in global projects?
a. Time zone utilization
b. Non-negotiable requirements
c. Static teams
d. Meeting notes and documentation are easy accessible

Quiz 17: Agile Architecture

Question: How are estimations done based on story mapping?
a. The first estimate of the potential development team will be tracked against reality and every deviation will be managed individually
b. After the first few sprints, the backlog is re-estimated and the assumed velocity is checked against the real velocity
c. The potential development team quickly estimates all known epics/tasks relatively, and then has a closer look at 2-5 user stories
d. The most senior developer is doing the first estimate and discusses everything with the project manager
Question: What is correct about user story mapping?
a. Tasks on the right side have to be done before tasks on the left side
b. Tasks on top are more important than tasks on the bottom
c. To define tasks for releases, the tasks are split horizontally into groups
d. We do not focus on users, but only on tasks and activities they perform

Question: What are typical guidelines about architectural decisions?
a. In an agile project, software architecture should not be documented, because changes occur too often
b. Political and organizational constraints influence the software architecture
c. Decisions should be delayed until the last responsible moment (and that might be the start of the project/product development)
d. Software architecture must always be done in detail upfront
Question: What can be said in general about software architecture?
a. Arc42 is a standard for architecture documentation that focuses on different views on the system

b. A good architecture is something that only software architects and software developers understand
c. Technical layering is more important than domain layering in the long run
d. Software architecture is something that evolves over time and has to be managed carefully
Question: Why are quality attributes so important?
a. Quality attributes are usually distributed over the system, so they have to be taken into account on most changes of the system
b. Quality attributes, also known as quality goals or non-functional requirements, often conflict with each other and therefore have to be balanced with knowledge of the domain of the system
c. All quality goals are equally important and therefore are the same for all systems
d. Quality attributes define the detailed functional aspects of a system and therefore define the business logic

Quiz 18: Release Management

Question: What are best practices for release management?
a. The release process needs to be reliable and repeatable
b. Release externally first and then promote the release to internal test users
c. Automate every step of the build process
d. Release early and often to obtain user feedback
Question: Select the correct terminology
a. Continuous integration: Technique where members of a team integrate their work frequently.
b. Continuous delivery: Approach in which teams keep producing valuable software in short cycles and ensure that the software can be reliably released at any time.
c. Continuous deployment: Every change that passes automated tests is automatically deployed.
d. Continuous Software Engineering: Organizational capability to develop, release and learn from software in short cycles.
Question: What is meant by DevOps?
a. An approach that can only be used by small companies
b. An acronym for Development + Operations
c. An emphasis on development, quality and operation departments to work closely together to release software faster and more reliable
d. A new software process model

Quiz 19: Risk and Demo Management

Question: What is risk? a. A conflict b. A potential harm c. A problem d. A deviation from an expectation **Question:** What are dimensions of the risk assessment matrix? a. Impact b. Control c. Mitigation d. Probability **Question:** What are typical risk mitigation strategies? a. Risk ignorance b. Risk transfer c. Risk control d. Risk avoidance **Question:** What is described by Murphy's Law? a. If there's more than one way to do a job and one of those ways will end in disaster, then somebody will do it that way. b. When things are going well, something will go wrong c. Nothing can go wrong d. Anything that can go wrong will go wrong **Question:** What are components of a formalized scenario? a. Conditions that have to be met before the scenario starts b. Schedule that describes the time plan for the implementation of the scenario c. Interaction between actors and system d. Top level design of the architecture of the scenario **Question:** What are activities during the textual analysis in Abbot's Technique? a. Identify verbs as candidates for classes b. Identify nouns as candidates for methods c. Identify verbs as candidates for methods d. Identify nouns as candidates for classes