Medical Software Engineering

Successful execution of software projects
Version 3.0

Ronald Tanner, David Herzig

Version History

Version	Author	Description	Date
1.0	R. Tanner	initial version	2019/09/10
2.0	D. Herzig	MSc Medical Informatics	2020/03/13
3.0	D.Herzig	MSc Medical Informatics	2022/02/21

1 Course overview

1.1 Learning goals

- Systematic approach to gain knowledge in theories, methods, and tools to design and build a software that meets the specifications efficiently, cost-effectively, and ensures quality.

1.2 Content

- Introduction
- Software project planning
- Software development life cycle (SDLC):
  - Sequential, iterative and agile models
- Software Requirements Analysis
- Software Design:
  - Design patterns
  - Use Cases: Docker, JDBC, Hibernate, Spring
  - Unit-Testing
- Configuration management
  - Git
- Build Tools
  - Ant, Maven
- Testing
  - Code analyzer, logging
  - Unit tests
  - Performance tests, memory tests, profiling
  - User interface tests

1.3 Certificate of achivement

The final grade will be calculated based on two inputs:

- 30% Course project
- 70% Written exam

1.4 Timetable

TO BE DEFINED	TO BE DEFINED

1.5 Required Software Environment

The main topic of this course is software engineering and how to build high quality software systems. The complete course does not depend on any specific platforms nor technologies. To fulfill the exercises it is (of course) needed to use some specific technologies. These technologies could be used on any operating system (e.g. Windows, OSX, Linux). In the following section there is an overview of these technologies and a short description.

- **Java Development Kit 15/16/17** (https://www.oracle.com/ch-de/java/technologies/javase-downloads.html)
- IDE Integrated Development Environment There are 3 very popular IDEs:
  - **Eclipse** (https://www.eclipse.org/)
  - **IntelliJ** (https://www.jetbrains.com/de-de/idea/)
  - **Netbeans** (https://netbeans.org/)

  All of these environments have their pros and cons. Within this course, IntelliJ will be used
- **Atom** (https://atom.io/) A simple text editor is one of the most powerful tools. Atom will be used, as this editor is available on most common operating systems.
- **Apache ANT** (https://ant.apache.org/) Build Tool
- **Apache Maven** (https://maven.apache.org/) Build Tool
- **Apache Tomcat** (http://tomcat.apache.org/) Simple Servlet Engine
- **Apache JMeter** (https://jmeter.apache.org/) Test Tool (Load Tests)
- **Gradle** (https://gradle.org/) Build Tool
- **Git** (https://git-scm.com/) Source Code management
- **MySQL** (https://www.mysql.com/) A relational database
- **StarUML** (https://staruml.io/) Graphical UML editor

There will be several other frameworks which will be used during the course. These will be installed on demand. The ones listed above should be installed prior the course start.

2 Introduction and Overview

2.1 Current Situation and Challenges

- Software is ubiquitous and essential.
- Software is expensive: operation and maintenance costs may increase development costs by factors.
- Software development is risky: high failure and cancellation rate, frequent budget overrun and delays on schedule, many defective, inadequate products posing serious security threats.

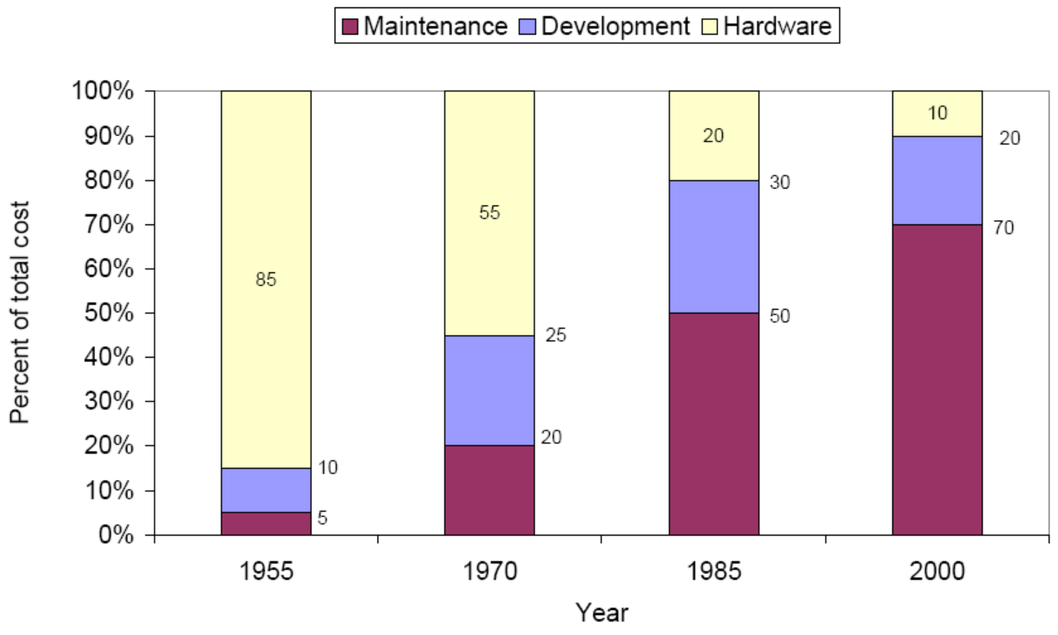
  Examples www.tricentis.com/blog/real-life-examples-of-software-development-failures

  Top Risks are:

  - Schedule (e.g. Time estimation)
  - Budget (e.g. Cost overrun)
  - Operational and Management (e.g. Resource planning)
  - Technical (e.g. Change of requirements)
  - External (e.g. Government rule change)
- The complexity and feature range of software applications is impressive and still rapidly increasing.

2.1.1 Software Maintenance Cost

Boehm ´80, Lientz ´87, US DoD ´97



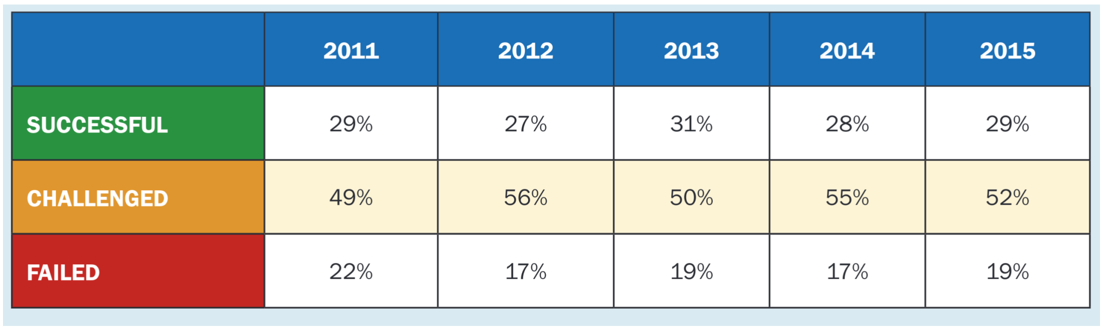
2.1.2 Learning from Failures?

Alfred Spector (1986)

> Bridges are normally built on-time, on-budget, and do not fall down. On the other hand, software never comes in on-time or on-budget. In addition, it always breaks down...When a bridge falls down, it is investigated and a report is written on the cause of the failure. This is not so in the computer industry where failures are covered up, ignored, and/or rationalised. As a result, we keep making the same mistakes over and over again.

2.1.3 Standish Group Reports

Since 1994 the Standish Group investigates software projects and publishes their findings:



	2011	2012	2013	2014	2015
SUCCESSFUL	29%	27%	31%	28%	29%
CHALLENGED	49%	56%	50%	55%	52%
FAILED	22%	17%	19%	17%	19%

SUCCESSFUL PROJECTS	% OF RESPONSES	CHALLENGED PROJECTS	% OF RESPONSES
User involvement	15.9	Lack of user input	12.8
Executive mgmnt support	13.9	Incomplete requirements	12.3
Clear statement of requirements	13.0	Changing requirements	11.8
Proper planning	9.6	Lack of executive support	7.5
Realistic expectations	8.2	Technology incompetence	7.0
Smaller project milestones	7.7	Lack of resources	6.4
Competent staff	7.2	Unrealistic expectations	5.9
Ownership	5.3	Unclear objectives	5.3
Clear vision and objectives	2.9	Unrealistic time frame	4.3
Hard-working, focused staff	2.4	New technology	3.7
Other	13.9	Other	23.0

Research at the Standish Group also indicates that **smaller time frames, with delivery of software components early and often**, will increase the success rate. Shorter time frames result in an iterative process of design, prototype, develop, test and deploy small elements. This process is known as growing software, as opposed to the old concept of developing software. Growing software **engages the user earlier**, each component has an owner or a small set of owners, and expectations are realistically set. In addition, each software component has a clear and precise statement and set of objectives. Software components and small projects tend to be less complex. Making the projects simpler as a worthwhile endeavor because complexity causes only confusion and increased cost.

2.2 Taking a Closer Look

IEEE Standard Computer Dictionary (Std 610):

> (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
>
> (2) The study of approaches as in (1)

Ian Sommerville:

> ...an engineering discipline that is concerned with all aspects of software production from initial conception to operation and maintenance.

B. Boehm (1981)

> is the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures and associated documentation

Software Engineering Body of Knowledge (SWEBOK V3):
www.computer.org/education/bodies-of-knowledge/software-engineering

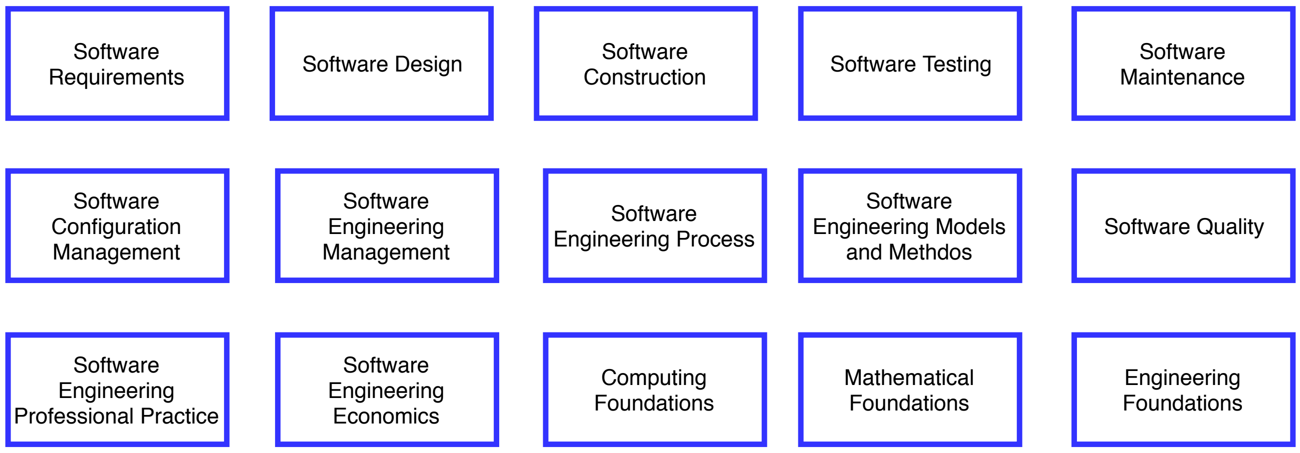

Software Requirements
Software Design
Software Construction
Software Testing
Software Maintenance
Software Configuration Management
Software Engineering Management
Software Engineering Process
Software Engineering Models and Methdos
Software Quality
Software Engineering Professional Practice
Software Engineering Economics
Computing Foundations
Mathematical Foundations
Engineering Foundations

Figure 2.1: 15 Knowledge Areas (KA)

- **Software Requirements**: elicitation, negotiation, analysis, specification, and validation,
- **Software Design**: definition of the architecture, components, interfaces,
- **Software construction**: detailed design, coding, unit testing, integration testing, debugging, verification and tools,
- **Software Testing**: fundamentals of software testing; testing techniques; human-computer user interface testing and evaluation; test-related measures
- **Software Maintenance**: program comprehension, re-engineering, reverse engineering, refactoring, software retirement; disaster recovery techniques, tools,
- **Software Engineering Professional Practice**: standards, codes of ethics; group dynamics (working in teams, cognitive problem complexity, interacting with stakeholders, dealing with uncertainty and ambiguity, dealing with multicultural environments); communication skills
- **Software Engineering Economics**: cost-benefit analysis, optimization, risk analysis, estimation, decision making
- **Software Configuration Management**: identification, control, status accounting, auditing; software release management and delivery, tools
- **Software Engineering Management**: process planning, estimation of effort, cost, and schedule, resource allocation, risk analysis, planning for quality; product acceptance; review and analysis of project performance; project closure; tools
- **Software Engineering Process**: software life cycle models, process assessment, tools
- **Software Engineering Methods**: modeling, analysis
- **Software quality**: verification, validation, reviews, audits, tools
- **Computing Foundations**: operation systems, networking, algorithms, parallel an distributed computing
- **Mathematical Foundations**: sets, relations, and functions; basic propositional and predicate logic; proof techniques; graphs and trees; discrete probability; grammars and finite state machines; and number theory.
- **Engineering Foundations**: statistical analysis; measurements and metrics; engineering design; simulation and modeling

7 Related Disciplines

- Computer Engineering
- Computer Science
- General Management
- Mathematics
- Project Management
- Quality Management
- Systems Engineering

2.2.1 Craft and Pragmatism

Is software development more a craft than an **engineering** discipline? (Jack W. Reeves, 1992)

Pragmatism considers words and thought as tools and instruments for prediction, problem solving and action, and rejects the idea that the function of thought is to describe, represent, or mirror reality.

Pragmatists contend that most philosophical topics—such as the nature of knowledge, language, concepts, meaning, belief, and science—are all best viewed in terms of their practical uses and successes. (en.wikipedia.org/wiki/Pragmatism)

Examples from The Pragmatic Programmer

blog.codinghorror.com/a-pragmatic-quick-reference

- **Don't Gather Requirements – Dig for Them** Requirements rarely lie on the surface. They're buried deep beneath layers of assumptions, misconceptions, and politics.
- **Work With a User to Think Like a User** It's the best way to gain insight into how the system will really be used.
- **Program Close to the Problem Domain** Design and code in your user's language.
- **Make Quality a Requirements Issue** Involve your users in determining the project's real quality requirements.
- **There Are No Final Decisions** No decision is cast in stone. Instead, consider each as being written in the sand at the beach, and plan for change.
- **Some Things Are Better Done than Described** Don't fall into the specification spiral – at some point you need to start coding.
- **Don't Be a Slave to Formal Methods** Don't blindly adopt any technique without putting it into the context of your development practices and capabilities.
- **Critically Analyze What You Read and Hear** Don't be swayed by vendors, media hype, or dogma. Analyze information in terms of you and your project.
- **You Can't Write Perfect Software** Software can't be perfect. Protect your code and users from the inevitable errors.
- **Don't Live with Broken Windows** Fix bad designs, wrong decisions, and poor code when you see them.
- **DRY – Don't Repeat Yourself** Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.
- **Use the Power of Command Shells** Use the shell when graphical user interfaces don't cut it.
- **Use a Single Editor Well** The editor should be an extension of your hand; make sure your editor is configurable, extensible, and programmable.
- **Always Use Source Code Control** Source code control is a time machine for your work – you can go back.
- **Minimize Coupling Between Modules** Avoid coupling by writing "shy" code and applying the Law of Demeter.

- **Refactor Early, Refactor Often** Just as you might weed and rearrange a garden, rewrite, rework, and re-architect code when it needs it. Fix the root of the problem.

  Boy Scout Principle: Leave the campground cleaner than you found it.
- **Design to Test** Start thinking about testing before you write a line of code.
- **Test Early. Test Often. Test Automatically** Tests that run with every build are much more effective than test plans that sit on a shelf.
- **Don't Use Manual Procedures** A shell script or batch file will execute the same instructions, in the same order, time after time.
- **Sign Your Work** Craftsmen of an earlier age were proud to sign their work. You should be, too.

2.2.2 Software Application Types

No size fits all: appropriate engineering methods and techniques depend on the type of application:

1. Stand-alone applications
2. Interactive transaction-based applications
3. Embedded control systems
4. Batch processing systems
5. Entertainment systems
6. Systems for modeling and simulation
7. Data collection and analysis systems
8. Systems of systems

Source: Ian Sommerville, Software Engineering

2.2.3 A Simplified Model

Software Development

Requirements: elicitation, negotiation, analysis, specification, and validation,

Design: architecture, components, interfaces,

Construction: coding, unit & integration testing, documentation

Project Management:

Configuration Management: change control, release management

Process Life Cycle: planning, resource allocation, risk analysis

Quality Management: verification, validation, reviews

2.3 Sample Software Projects

2.3.1 Ariane 5

On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its lift-off from Kourou, French Guiana. The rocket was on its first voyage, after a decade of development costing $7 billion. The destroyed rocket and its cargo were valued at $500 million. A board of inquiry investigated the causes of the explosion and in two weeks issued a report. It turned out that the cause of the failure was a software error in the inertial reference system (IRS) untested for use in a new launch environment. Specifically a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer. The number was larger than 32767, the largest integer storeable in a 16 bit signed integer, and thus the conversion failed. (http://www-users.math.umn.edu/ arnold/disasters/ariane.html)

2.4 Software Engineering in health care and biomedicine

Software in health care is incredibly diverse. There are many different working environments:

- Creating and managing software systems that connect researchers with clinical experiment data (e.g. clinical trial)
- Developing mobile application to monitor and collect data of patients (e.g. collecting data with a smart watch).
- Providing health care providers with data records about a patient (e.g. doctor appointment).
- Implement software systems which allows the handling of big data sets (e.g. creating new software to predict skin cancer based on historical data).
- Implement software systems for embedded devices (e.g. software for a ventilator)).
- Implement software systems to support any health care environment (e.g. electronic lab journal).
- ... and many more

There are several aspects which must be considered:

- Data Security: handling of patient data
- Reliability: devices are responsible for the health of patients (e.g. a ventilator)
- Traceability: Each change in a system must be documented (e.g. which input data and source code was used to create the output data)

2.5 Exercises

1. What are the basic tasks that all software engineering projects must handle?
2. List five tasks that might be part of software deployment and explain why.
3. Collect several reasons why software engineering projects could fail.

3 Coursework

3.1 Introduction

A project will cover most of the topics from this course. At the end of each chapter, there will be a set of exercises which are used to strengthen the knowledge in software engineering. In addition, each student will have the possibility to play around with the tools demonstrated in this course.
For all the coursework, each student needs to have a computer (Windows, OSX or Linux) with the possibility to install additional software.

3.1.1 Projects

There are two projects available. The two projects are developed for the following two courses:

- Gene Information Service (FHNW - Master Medical Informatics)
- Raspberry Pi GPIO Web Application (FHNW - Master Automation Management)

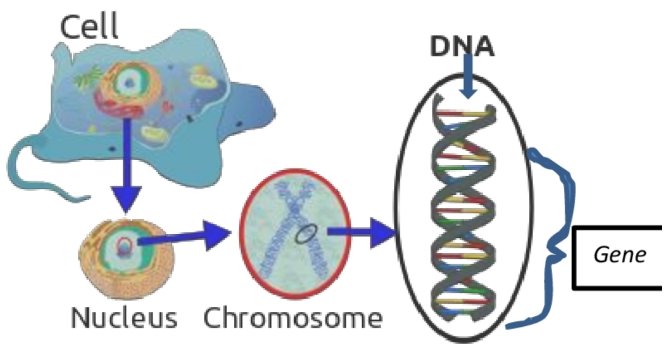
3.2 Gene Information Service

The gene information service will be a system with three components:

- **Data Loader** This component loads a data file from NCBI into a database
- **Gene Information Service** This component will provide an API which could be used by other applications to retrieve gene specific information
- **Gene Search Web Application** This component will provide a simple Java based Web Application which utilizes the Gene Information Service

3.3 What is a gene

In biology, a gene is a sequence of nucleotides in DNA or RNA that encodes the synthesis of a gene product, either RNA or protein. (Source: Gene: Wikipedia)



3.4 Basic Idea

The gene information service will provide a system to retrieve information for a given gene.

- The data which is used is coming from NCBI. The raw data size is in the area of 4GB.
- This data will be loaded into a database (e.g. MySQL, Postgres). This will be done by the first component, *Data Loader*
- The data will be available over a REST interface. This is the second component. The service will be connected to the database a will provide query functionalities for end users.
- A end user application will be created, where potential users could create queries and display the gene information data. This could be a web application or also a mobile application. In this project, we will use a web application.

3.5 Exercises

1. In each project it is important to understand the data which is used. Download the gene information file from the NCBI ftp server.

   `ftp://ftp.ncbi.nlm.nih.gov/gene/DATA/gene_info.gz`

   Try to answer the following questions:

   - How many genes does the file contain?
   - What kind of information is in the tax id field?
   - Which gene types do exist?
   - Which gene type is the one which appears the most?

4 Project Planning

This chapter will cover methods how to control a project. Stakeholders measure projects by how well they are executed within the project constraints or baselines. A project baseline is an approved plan for a portion of a project. It is used to compare actual performance to planned performance and to determine if project performance is within acceptable guidelines. The following 4 project baselines should be considered:

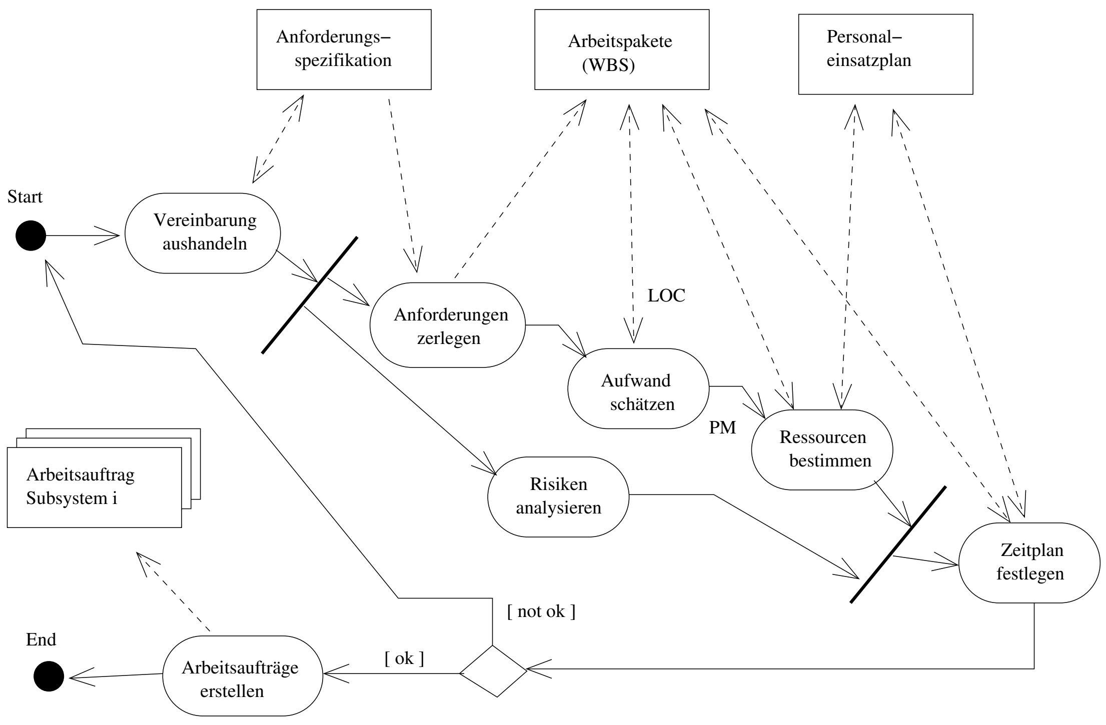
- Quality
- Schedule
- Scope
- Budget

Results:

- **C**ost plan: Creating budget / offer.
- **T**ime plan: Define activities, deadlines and milestones.
- **E**mployee plan: Nomination of people and their working time.
- **O**rganization plan: Agreement of team structure, nomination of responsible persons.
- **Q**uality plan: Compilation of documents, tools and methods to ensure quality.
- **P**roject monitor plan: Agreement of actions to control the project and if needed how to change the plans.
- **C**onfiguration management plan: Agreement of actions to control changes in documents, code or data (or any other resources).
- **E**ducation plan: Definition of education for internal (project team) and external people (users, operating stuff).
- **R**isk management plan: Agreement of actions to avoid or mitigate risks.

4.1 Classical Project Project Planning

The classic project planning approach contains the following activities:



1. **Arrange agreement**: Define the project scope with a high level time schedule, point in time of deliverables, definition of mile stones
2. **Breakdown requirements**: Breakdown the requirements into work packages according to the defined deliverables, create WBS (work breakdown structure). Recommended time for one work package is 4-6 weeks.
3. **Estimate effort**: Estimate effort for each package based on a value which could be measured. For software components, this could be LOC (lines of code). The LOC value could be used to get the amount of time needed for that component.
4. **Define resources**: Assign people to activities with respect of their availability, qualification (education) and demand. The best way to do that is the usage of a model with roles and responsibilities.
5. **Analyse risks**: Identify and prioritize potential risks and possible actions.
6. **Define schedule**: Description of all work packages and their time dependencies. When is the final point in time to deliver a package, are there any dependencies to other packages? This is typically done with a gantt chart or network diagram.

4.2 Agile Project Planning

Software development is a constant dialog between *What is possible* and *What is needed*.

- **Planning**: only plan for the next iteration, version. Not more!
- **Responsibility** will be taken over, not assigned
- **Estimates**: the responsible persons define the effort and duration
- **Priorities** define the order,

The business side (Product Owner) decides

- **MVP**: Minimal viable product. How should the problem be solved to bring additional value. What is to much, what is not enough?
- **Priority**: Which tasks should be executed first?
- **Release**: Which pieces should be part in the next release?
- **Delivery**: Which timeframe is available to be successful?

The developers have the freedom to take over responsibility and decide for

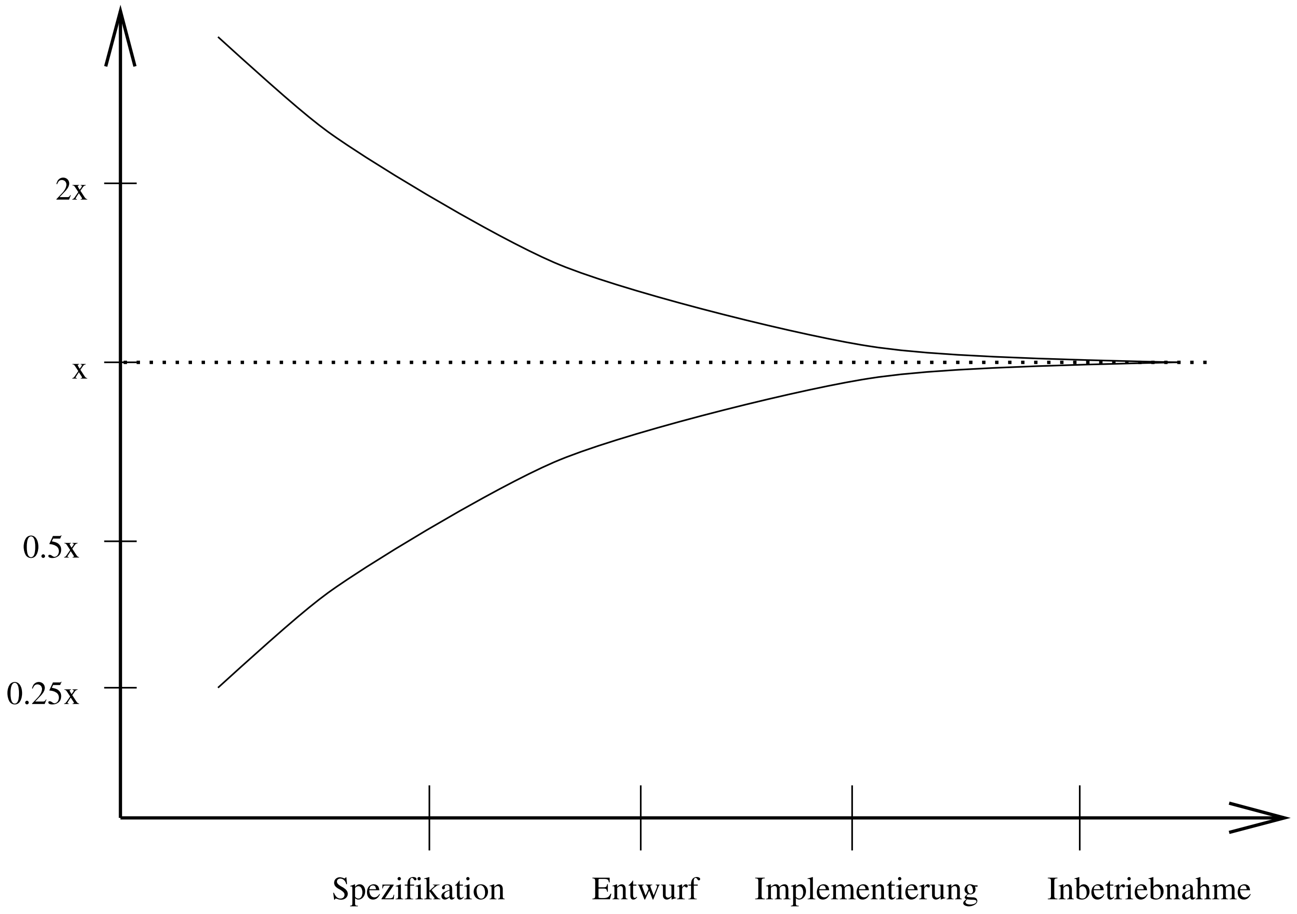
- **Effort**: how long will it take to implement a specific feature?
- **Consequences**: what are the consequences for a taken approach?
- **Process**: what is the structure for the work and the team?
- **Planning**: when will the features be delivered

4.3 Cost Estimation

The total costs of software projects contains

- **S**oftware-Cost: software, licenses
- **H**ardware-Cost: infrastructure (own hardware, cloud-based systems)
- **P**ersonal-Cost: Salary, Expenses

Cost Estimation Accuracy:



Influencing factors to estimate the costs:

- **S**ize: Lines of code, classes, methods
- **C**omplexity: Dependency between the modules, is it possible to build modules?, HW/SW environment
- **E**xperience: Number of similar executed projects
- **R**eliability: error rate, system stability

There are several methods to estimate the costs:

- Empiric estimation methods:
  - **Expert judgment:** Several experts on the proposed software development techniques and the application domain are consulted. They each estimate the project cost. These estimates are compared and discussed. The estimation process iterates until an agreed estimate is reached.
  - **Delphi method:** Delphi technique is quite an old but efficient forecasting method. It follows an interactive approach which relies on exchange of ideas. The team is composed of a group of experts in their respective domains, who answers the queries in two or more rounds. Every time a facilitator provides a summary of the collected ideas, which is revised by the experts if required. The process of opinion and revaluation goes on until a final consensus is reached. Delphi technique relies its assumption on the fact that assimilation of ideas from a structured group leads to a productive outcome.
- Algoritmic estimation methods:
  - **Function Point Analysis:** Function Point Analysis is a standardized method used commonly as an estimation technique in software engineering. In simple words, FPA is a technique used to measure software requirements based on the different functions that the requirement can be split into. Each function is assigned with some points based on the FPA rules and then these points are summarized using the FPA formula. The final figure shows the total man-hours required to achieve the complete requirement.
  - **Widget Point Analysis:** Count the UI (user interface) elements and calculate out of this number the function points. Only useful for software with user interfaces and not many algorithmic calculations in the background.
  - **Lines of Code (LOC):** Easy measurement method. Dependent on the programming language and the available libraries.
- Other methods:
  - **Pricing to win** : The software cost is estimated to be whatever the customer has available to spend on the project. The estimated effort depends on the customer's budget and not on the software functionality.
  - **Pain threshold method**: Highest price which the customer is willing to pay
  - **Parkinson's Law**: Parkinson's Law states that work expands to fill the time available. The cost is determined by available resources rather than by objective assessment. If the software has to be delivered in 12 months and 5 people are available, the effort required is estimated to be 60 personmonths.

Function Points (FP) (A.J. Albrecht) Function points are used to compute a functional size measurement (FSM) of software.: (www.ifpug.org)

Category	Number	Value (low) average (high)	Total
External inputs		x (3) 4 (6)	
External outputs		x (4) 5 (7)	
External inquiries		x (3) 4 (6)	
Internal logical files		x (7) 10 (15)	
External interface files		x (5) 7 (10)	
Total			

In a second step, the calculated function points could be converted into LOC (lines of code):

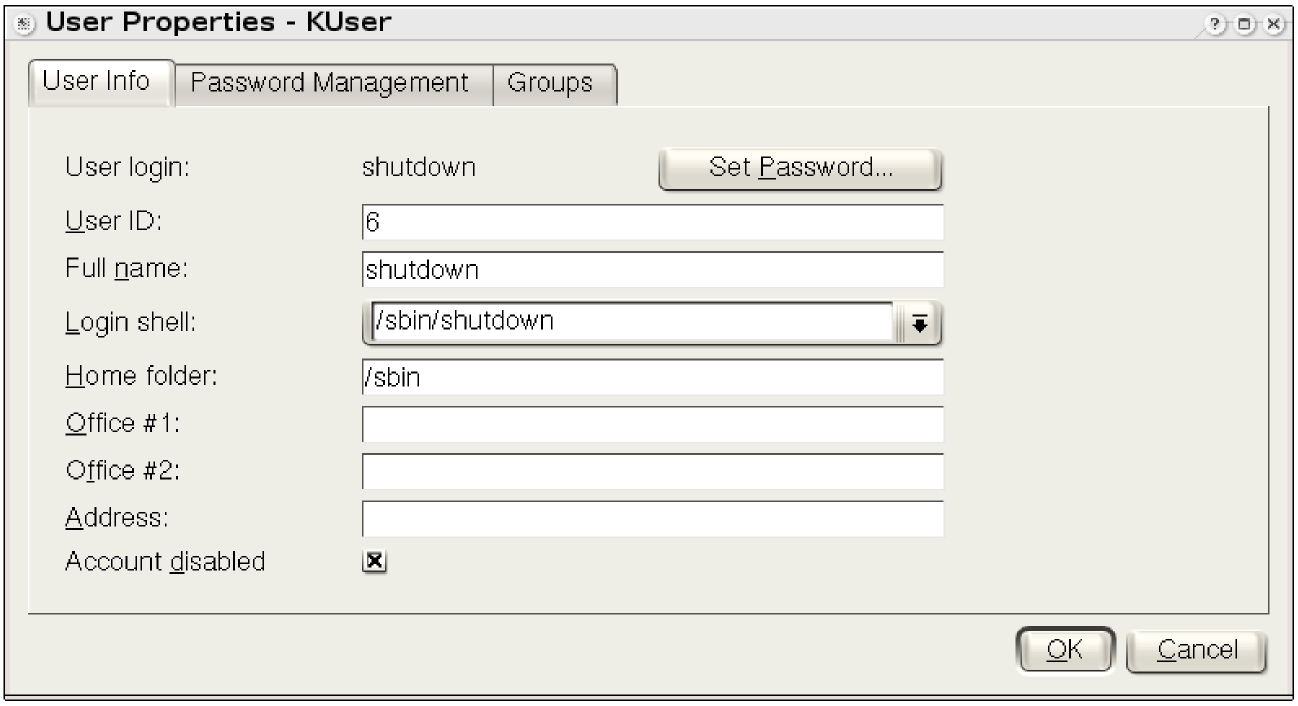
Programming language	LOC per FP
Assembler	320
C	128
Fortran	128
Pascal	91
C++/Java	53
SQL	13

Attention: this method does not reflect the re-usage of software components!

Widget-Points (H. Krasemann)[2ex] Calculate the number of function points based on the elements of the user interface.

$$functionpoints = 2 \cdot widgetpoints \tag{4.1}$$

- **Input widgets:** Textfield, Combobox, Menubutton, Radiobutton, Pushbutton, Checkbox
- **Describing widgets:** Label, Separator, Group box, Window
- **Composite widgets:** Notebook, Table, Scrollbar, List
- **Menu widgets:** Menu bars, Men items



User login:	shutdown
User ID:	6
Full name:	shutdown
Login shell:	/sbin/shutdown
Home folder:	/sbin
Office #1:	
Office #2:	
Address:	
Account disabled	☒

COCOMO (COnstructive-COst-MOdel, B. Boehm)

1. Calculation of the effort in person month:

$$E_i = a \cdot KDL^b \tag{4.2}$$

with the factor a the exponent b and KDL as number of delivered lines of code (in thousand) (Kilo-delivered-lines).

Project type	a	b
organic (simple)	3.2	1.05
semi-detached (medium)	3.0	1.12
embedded (complex)	2.8	1.20

2. Calculation of the effort in person month:

$$E = EAF \cdot E_i \tag{4.3}$$

with the correction factor EAF (Effort-adjustement-factor).

3. Possible correction factors EAF from the table:

Cost factors	very low	low	nominal	high	very high
Required Software Reliability	0.75	0.88	1.00	1.15	1.40
Size of Application Database		0.94	1.00	1.08	1.16
Complexity of The Product	0.70	0.85	1.00	1.15	1.30
Runtime Performance Constraints			1.00	1.11	1.30
Memory Constraints			1.00	1.06	1.21
Response times		0.87	1.00	1.07	1.15
Analyst capability	1.46	1.19	1.00	0.86	0.71
Applications experience	1.29	1.13	1.00	0.91	0.82
Software engineer capability	1.42	1.17	1.00	0.86	0.70
Programming language experience	1.14	1.07	1.00	0.95	
Application of software engineering methods	1.24	1.10	1.00	0.91	0.82
Use of software tools	1.24	1.10	1.00	0.91	0.83
Required development schedule	1.23	1.08	1.00	1.04	1.10

4. Calculation of the effort distribution (in percentage):

Software size	small (2 KDL)	medium (32 KDL)	large (128 KDL)
Design	16	16	16
Implementation	68	62	59
Integration + Tests	16	22	25

SCRUM: Story-Points

Story points are a unit of measure for expressing an estimate of the overall effort that will be required to fully implement a product backlog item or any other piece of work. When we estimate with story points, we assign a point value to each item. The raw values we assign are unimportant.

The Fibonacci sequence is one popular scoring scale for estimating agile story points. In this sequence, each number is the sum of the previous two in the series.

0, 1, 2, 3, 5, 8, 13, 21....

Also, it's useful to set a maximum limit (13, for instance). If a task is estimated to be greater than that limit, it should be split into smaller items. Similarly, if a task is smaller than 1, it should be incorporated into another task.

Before each estimation round, set 2 reference points: two good predictable stories, one with 2 the other one with 5 points.

Burn-Down Chart

A burndown chart shows the amount of work that has been completed in an epic or sprint, and the total work remaining. Burndown charts are used to predict your team's likelihood of completing their work in the time available.

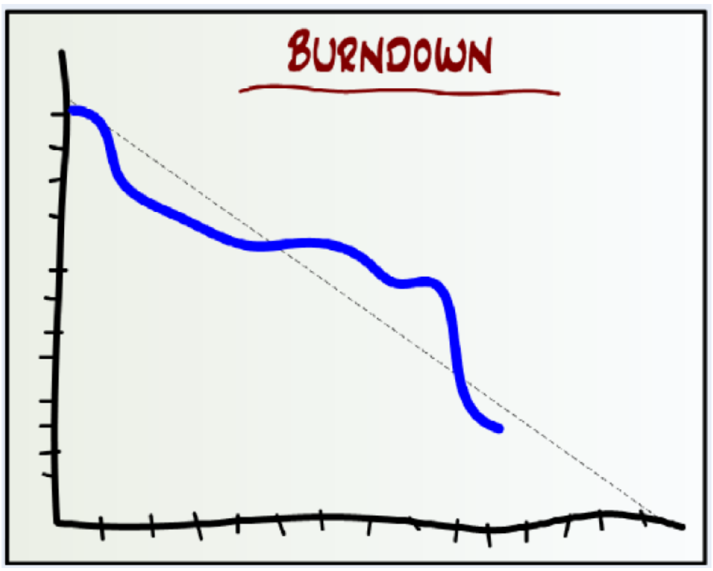


Figure 4.1: Burndown chart (Quelle: Henrik Kniberg)

And what about Microsoft Project?

Software development guru Joel Spolsky puts it this way:

> The trouble with Microsoft Project is that it assumes that you want to spend a lot of time worrying about dependencies... I've found that with software, the dependencies are so obvious that it's just not worth the effort to formally keep track of them. Another problem with Project is that it assumes that you're going to want to be able to press a little button and "rebalance" the schedule... For software, this just doesn't make sense [in practice]... The bottom line is that Project is designed for building office buildings, not software.

Joel is a former Microsoft employee, who worked on both Excel and Project.

- Cost estimation with Story Points (Mike Cohn) www.planningpoker.com/detail.html
- How to implement Scrum:

  www.agile-software-development.com/2007/09/how-to-implement-scrum-in-10-easy-steps_20.html

4.4 Timeplan

Project Schedule Plan

Axiom: persons and months are NOT exchangeable!

1. Calculate the length of a project in months based on the calculated person months:

$$D = 2.5 \cdot E^{\,0.38} \tag{4.4}$$

2. Caclculate the average demand of employees:

$$P = \frac{E}{D} \tag{4.5}$$

3. Calculation of the effort distribution (in percentage):

Software size	small (2 KDL)	medium (32 KDL)	large (128 KDL)
Design	19	19	19
Implementation	63	55	51
Integration + Tests	18	26	30

4. Refinement of all phases and creation of the predecessor list:

No	Task	Predecessor	Duration	Ressource
1	Software design		5	Meier
2	Design approval	1	1	(Review)
3	Implementation GUI	2	25	Hilfiker
4	Implementation DB	2	12	...
5	Module test GUI	3	3	...
6	Module test DB	4	2	...
7	Integration	5,6	2	...

Further development of the model:

sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html

Calculation software:

- COCOMO 81 Intermediate Model Implementation:

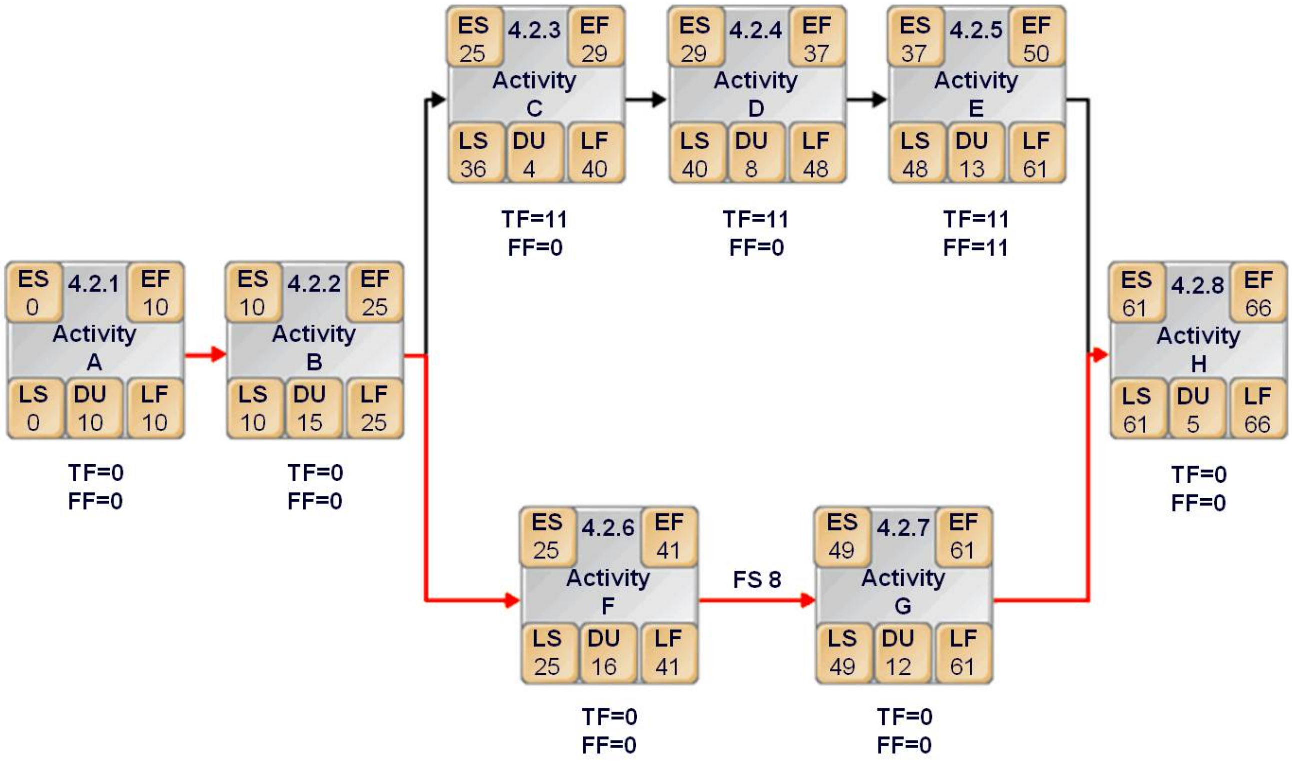
  sunset.usc.edu/research/COCOMOII/cocomo81_pgm/cocomo81.html

- COCOMO II with Heuristic Risk Assessment:

  sunset.usc.edu/research/COCOMOII/expert_cocomo/expert_cocomo2000.html

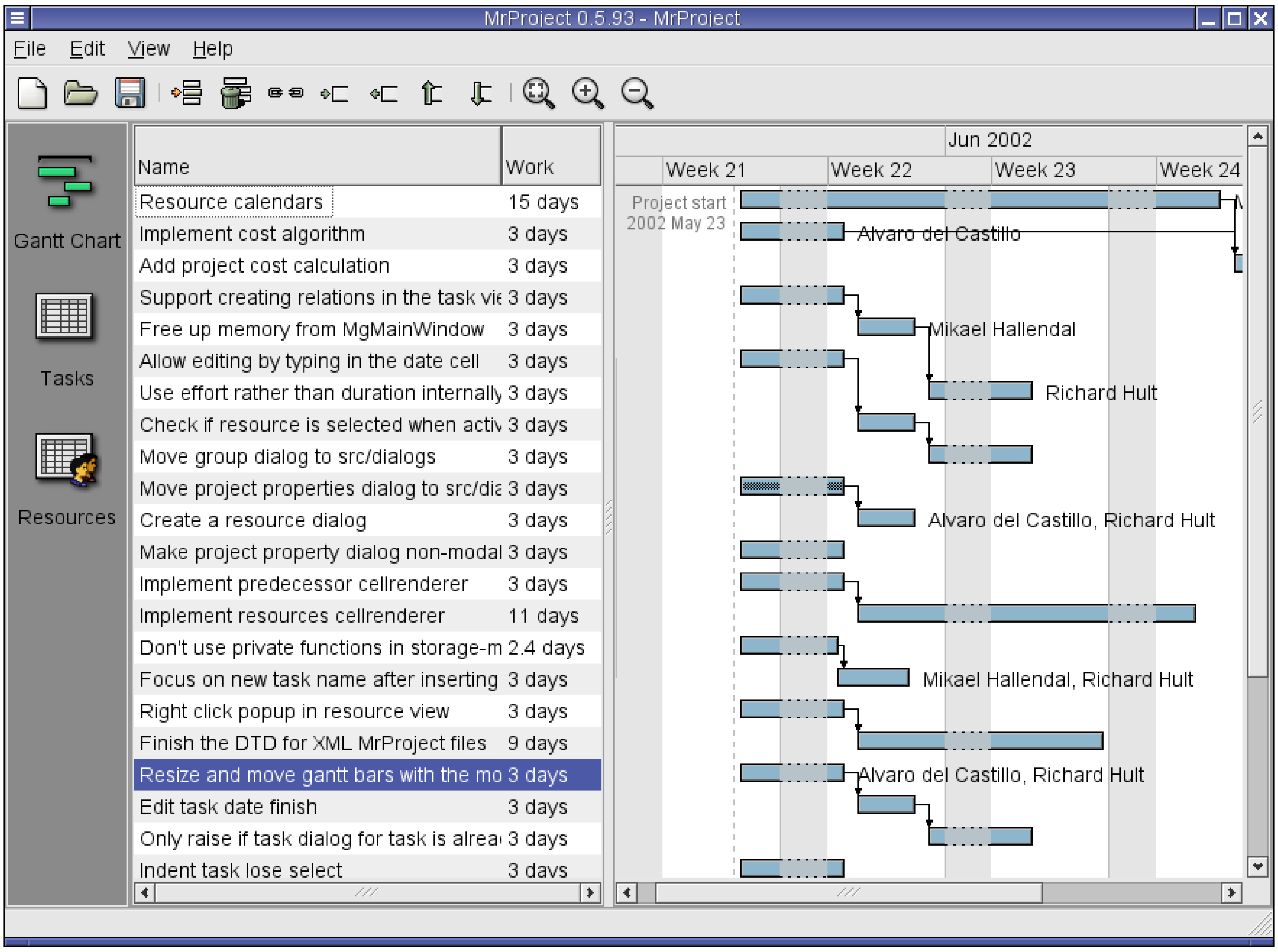
4.4.1 Critical Path Method / Critical Path Analysis

The critical path method (CPM), or critical path analysis (CPA), is an algorithm for scheduling a set of project activities. It is commonly used in conjunction with the program evaluation and review technique (PERT). A critical path is determined by identifying the longest stretch of dependent activities and measuring the time required to complete them from start to finish.


ES 25 4.2.3 EF 29 Activity C LS 36 DU 4 LF 40
TF=11 FF=0
ES 29 4.2.4 EF 37 Activity D LS 40 DU 8 LF 48
TF=11 FF=0
ES 37 4.2.5 EF 50 Activity E LS 48 DU 13 LF 61
TF=11 FF=11
ES 0 4.2.1 EF 10 Activity A LS 0 DU 10 LF 10
TF=0 FF=0
ES 10 4.2.2 EF 25 Activity B LS 10 DU 15 LF 25
TF=0 FF=0
ES 61 4.2.8 EF 66 Activity H LS 61 DU 5 LF 66
TF=0 FF=0
ES 25 4.2.6 EF 41 Activity F LS 25 DU 16 LF 41
TF=0 FF=0
FS 8
ES 49 4.2.7 EF 61 Activity G LS 49 DU 12 LF 61
TF=0 FF=0

4.4.2 Gantt-Chart

A Gantt chart is a type of bar chart that illustrates a project schedule. This chart lists the tasks to be performed on the vertical axis, and time intervals on the horizontal axis. The width of the horizontal bars in the graph shows the duration of each activity. Gantt charts illustrate the start and finish dates of the terminal elements and summary elements of a project. Terminal elements and summary elements constitute the work breakdown structure of the project. Modern Gantt charts also show the dependency relationships between activities. (wikipedia.org)



Name	Work
Resource calendars	15 days
Implement cost algorithm	3 days
Add project cost calculation	3 days
Support creating relations in the task vie	3 days
Free up memory from MgMainWindow	3 days
Allow editing by typing in the date cell	3 days
Use effort rather than duration internally	3 days
Check if resource is selected when activ	3 days
Move group dialog to src/dialogs	3 days
Move project properties dialog to src/dia	3 days
Create a resource dialog	3 days
Make project property dialog non-modal	3 days
Implement predecessor cellrenderer	3 days
Implement resources cellrenderer	11 days
Don't use private functions in storage-m	2.4 days
Focus on new task name after inserting	3 days
Right click popup in resource view	3 days
Finish the DTD for XML MrProject files	9 days
Resize and move gantt bars with the mo	3 days
Edit task date finish	3 days
Only raise if task dialog for task is alrea	3 days
Indent task lose select	3 days

Figure 4.2: Quelle: live.gnome.org/Planner

The length of a bar is the time which is used for a specified activity. Create the diagram in respect of using a meaningful level of detail.

4.5 Risk Management

Recognize, analyse and control risks:

Risk	Reason
Change of requirements	Scope variations occur when the scope of an iteration changes after a timeframe had been agreed upon. Due to the value from receiving frequent customer feedback, stakeholders or product owners will often ask to vary the scope of a project
Unrealistic planning	Inaccurate estimations occur when the length of a project, milestone or iteration is underestimated by the project group. Software estimations can cause problems between developers and clients because they lead to increase project timeframes, and therefore also project expenses. There could also be a problem if people are not available due to job change, illness, accident or other reasons.
Missing experience	first usage of a technology, method, tools, reduced knowledge of the project area, complex SW/HW environment, special requirements, big data, high reliability
Improper infrastructure	instable or non-working HW/SW components. Problem with component delivery by 3rd party vendor
Financial issue	Reorganization, incorrect budget estimation, project scope expansion, cost overruns
Communication issues	Inadequate support, misunderstanding, conflicts

Level of risk:

$$Risk = Likelihood \cdot Severity \tag{4.6}$$

Project management standards dictate that planning in advance for risks to the project is a critical success factors. Each risk should be identified and ranked on a scale of probability and severity (1-10 or similar) in a risk log.

Once prioritized, there are 5 primary ways to manage your project risks:

1. Avoidance Although often not possible, this is the easiest way of removing risk from a project. It involves the removal of the tasks that contain the risk from the project.
2. Acceptance On the other end of the spectrum, acceptance involves planning the risk into the project. If a better response strategy cannot be identified, accepting the risk might be sufficient to proceed with the project.
3. Monitor and Prepare Similar to accepting the risk, this response can be used for major risks that carry a high probability and/or severity, but must be accepted by the project. It involves the following two things:
   - Creating plans for monitoring the triggers that activate the risk.
   - Building action plans that can be immediately mobilized upon occurrence of the risk.
4. Mitigation Since risk is a function of probability and severity, both of these factors can be scrutinized to reduce the risk of project failure.
   - Probability of occurance: Take measures to reduce the likelihood of a risk occurring. This is usually a more preferable option than reducing the severity because it's better not to experience the risk occurrance in the first place.
   - Severity: Reduce the impact of the risk on the critical success factors of the project.

5. Transference Finally, you can transfer the risk onto another party. Naturally, this will usually require some form of trade-off (or cost). Shifting the consequence of a risk to a third party is not always easy but is often overlooked.

4.5.1 Risk Management: Everest Base Camp Trek

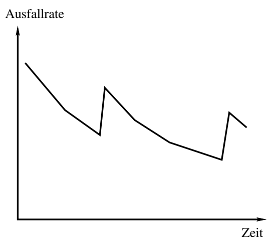
What are the risks on going to the Everest Base Camp? How could these risks are handled/managed?

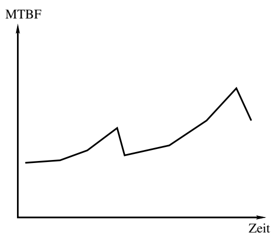
4.6 Measurements and Metrics

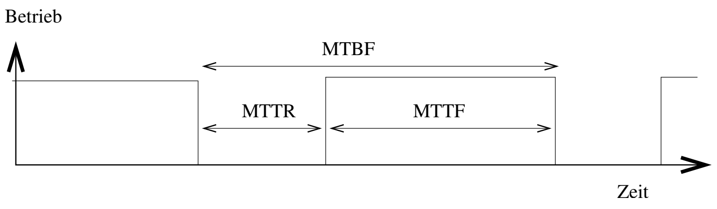
With measurements, attributes become quantifiable and measureable:

Attribute	Measurement
Software size	(LOC) Number of lines of code
Complexity	Number of classes, methods, relations
Change frequency	Number of changes in a given time frame
Fault rate	Number of errors in a given time frame
Productivity	Number of lines in a given time frame
Effort	Number of working hours

4.6.1 Failures, Bugs and Availability







failure
fault, bug
error
reliability
availability

MTBF	mean time between failures
MTTR	mean time to repair
MTTF	mean time to failure

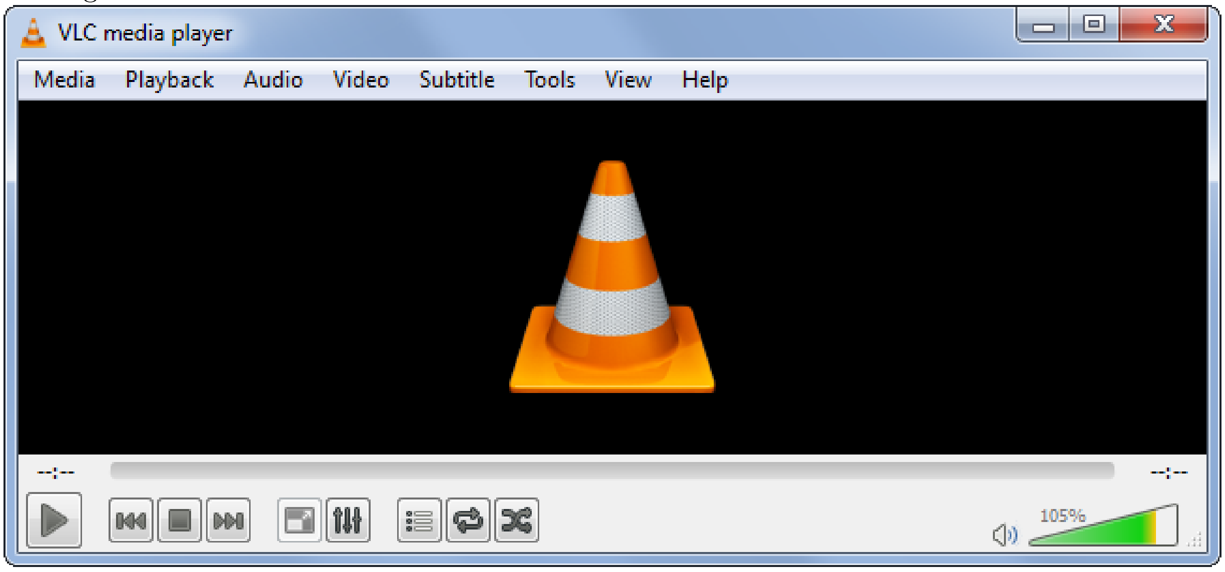
Nines of	downtime per year		
Reliability:	[h]	[min]	[sec]
2 9’s (99%)	87.6	5256.0	315360.0
3 9’s (99.9%)	8.76	525.6	31536.0
4 9’s (99.99%)	0.876	52.56	3153.6

4.7 Exercises

1. What is the critical path for the given task list and what is the duration (of the critical path)?

Task	Duration (Days)	Predecessor
A	4	- (Start)
B	3	A
C	1	B
D	5	- (Start)
E	4	D
F	6	E
G (end)	2	C,F
H	4	D
I (end)	2	H

2. VLC is a popular video player software. Try to find out how many lines of code are needed based on the given user interface:



3. You are responsible to find out how many person months are needed to create a simple time recording software. The requirements are the following:
   - The system is a single user system
   - A database is used to store the information
   - The user could add, delete or change an entry
   - An entry contains *date, time, task*
   - A report could be created for a given month. This report will list a activities for the specified month.

5 Software Life Cycle Models

The software development process consists of a set of related activities (tasks) with each having a start and an end date and producing some outcome.

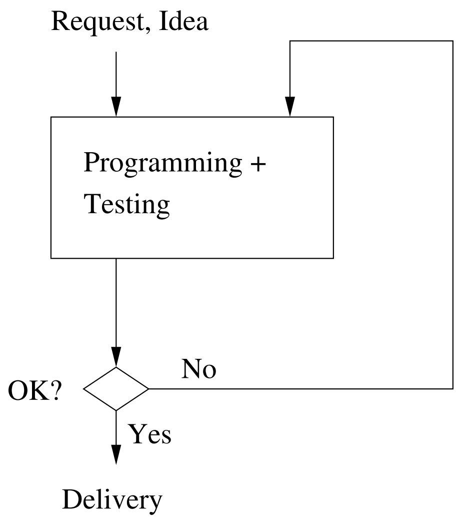
The software development life-cycle (SDLC) emphasizes the key activities and their temporal and logical interdependencies and relationships.

Common activities:

1. Specification: defining functionality and constraints
2. Development: building executable code
3. Validation: testing against requirements
4. Evolution: adapting to meet changing customer needs

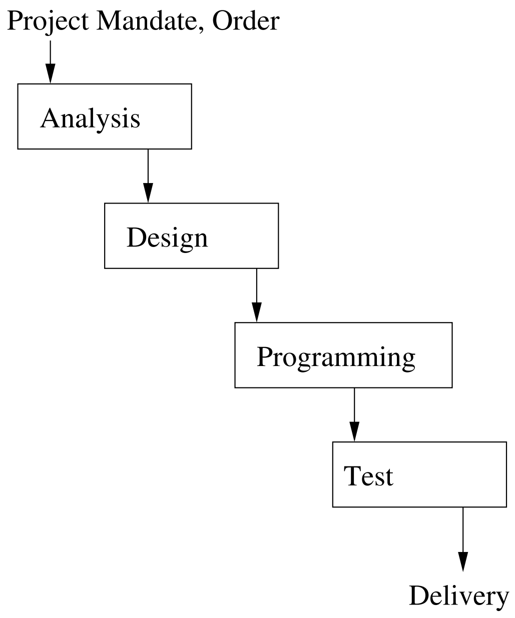
5.1 Adhoc Model

spontanous, individual process with little or no control:



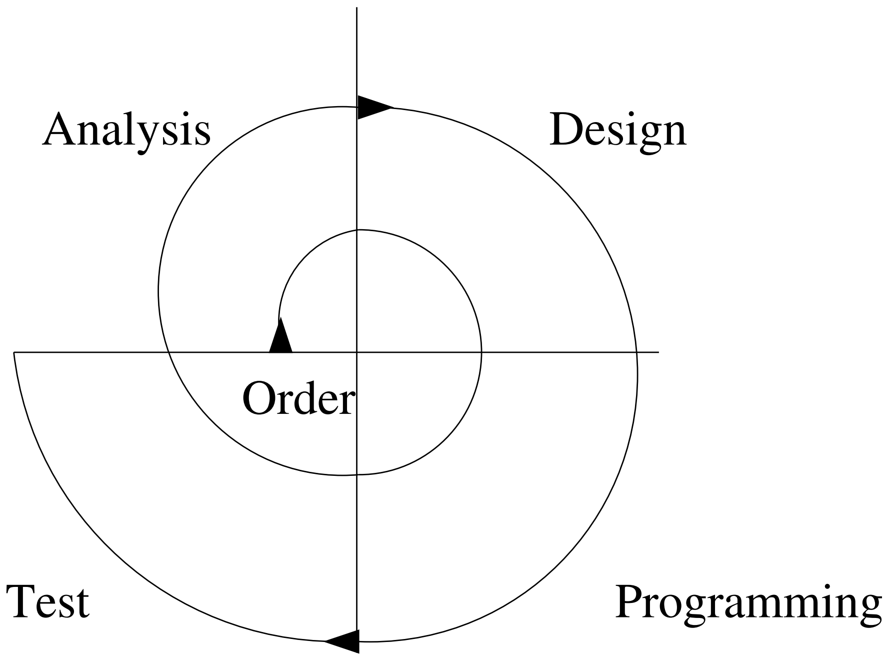
5.2 Waterfall Model

strictly sequential, immutable process



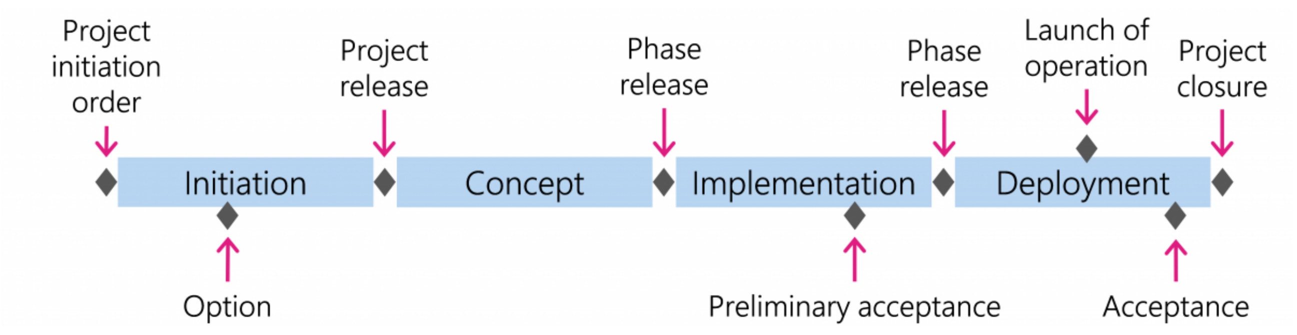
5.3 Spiral Model

cyclic, risk oriented



5.4 Hermes

An open standard for steering, managing and executing IT projects of service and product development and business organization within and outside of the Swiss Federal and cantonal administration (since 1975, latest update Version 5.1, 2014) www.hermes.admin.ch



Phases and milestones

Initiation definition of objectives and requirements with options to be developed and evaluated. The project order is created on the basis of the option selected.

Concept : detailed requirements specification and project-specific solution concepts, the system architecture with processes, functionality, system components and their integration into the system environment via interfaces.

Implementation system development and testing, decision on preliminary acceptance.

Deployment system activation and acceptance.

Szenarios: process adaption, customization to the project characteristics

- Customized IT application: development and integration,
- Standard IT application: procurement and integration,
- Service Product
- Organisation adjustment
- Individual scenario

Modules:

reusable building blocks for creating scenarios which contain the thematically related tasks, outcomes and roles

- Project steering
- Project management
- Agile development
- Project foundations
- Business organization
- Product
- IT system
- Procurement
- Deployment
- organization
- Testing
- IT migration
- IT operation
- Information security and data protection

Specific Modules: Marketing, Communication, Personnel development, Training, Strategy development, Business administration, Deployment

5.5 V-Model

Das V-Modell-XT an official project management method used by the German Government to provide guidance for planning and executing projects. www.v-modell-xt.de)

V-Model an US government standard and a general testing model.

V-Model-XT Key Concepts:

- Process specific adaption (Tailoring): Project Types, Project Modules and Work Products
- Decision Gate (Entscheidungspunkt): achievement of a process progress stage
- Project Types:
  - System Development Project of an Acquirer (Auftraggeber, AG)
  - System Development Project of a Supplier (Auftragnehmer, AN)
  - Introduction and Maintenance of an organization-specific process model

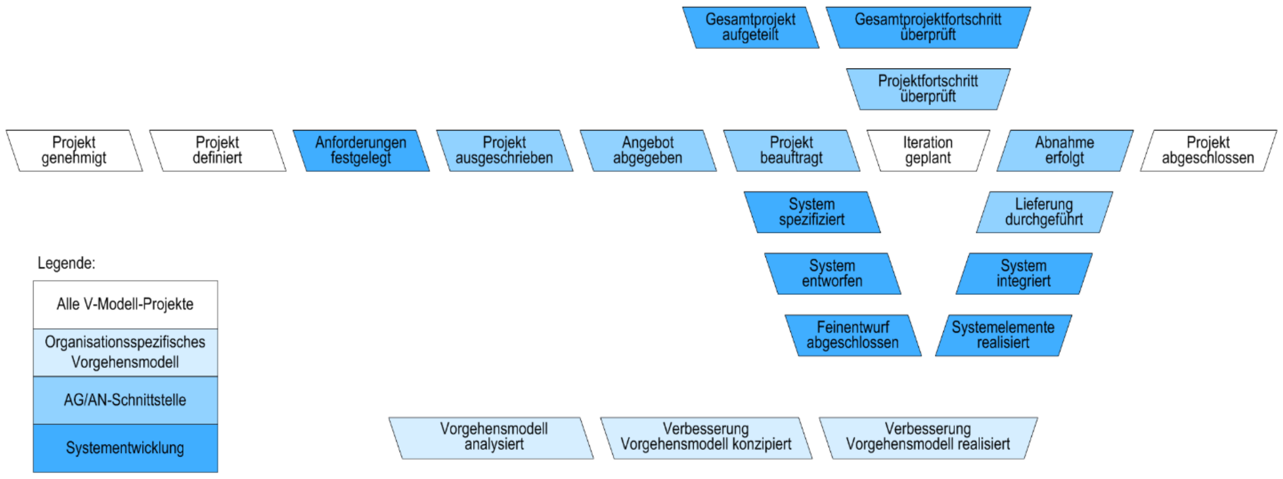


Figure 5.1: Project Execution Strategies

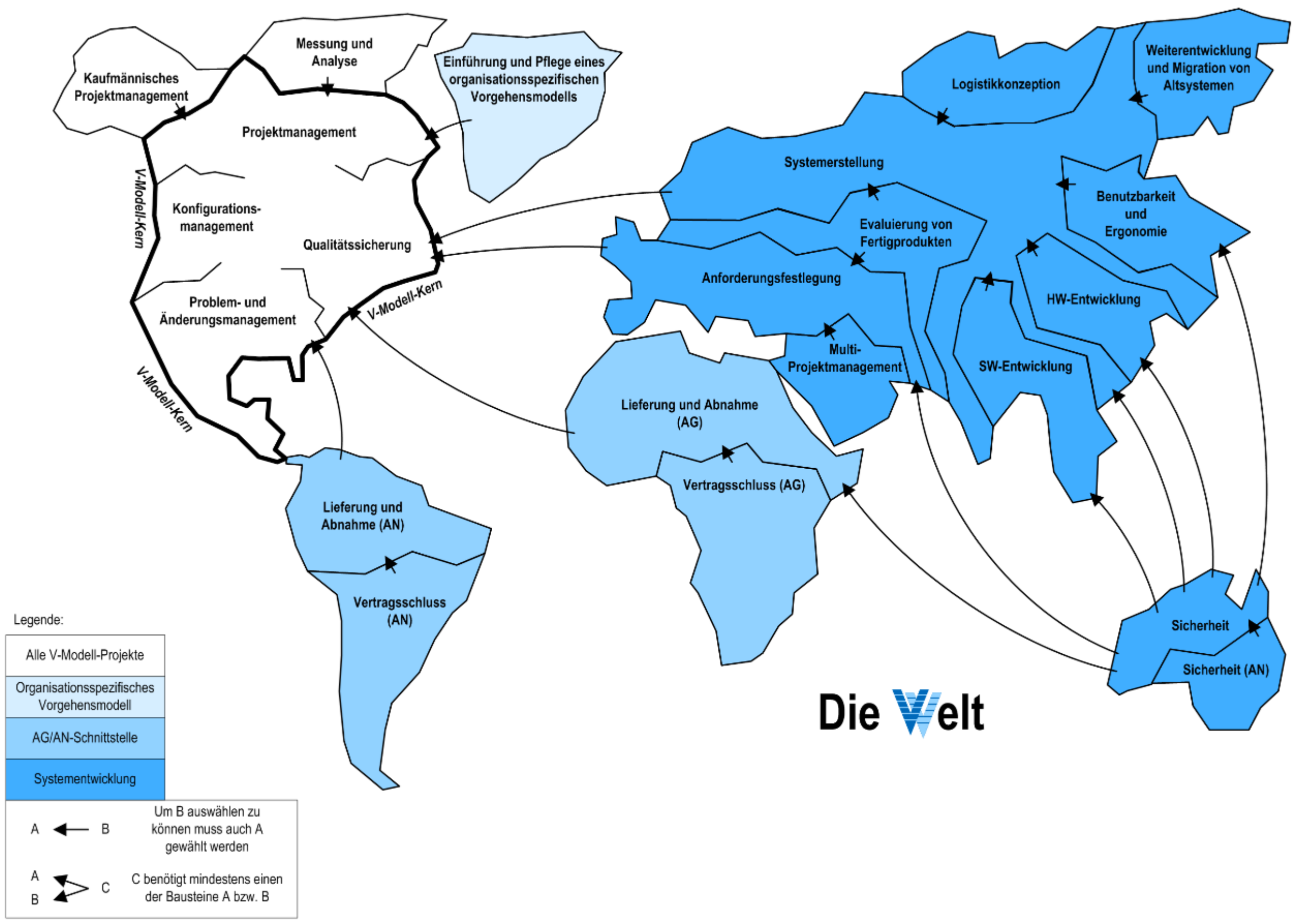
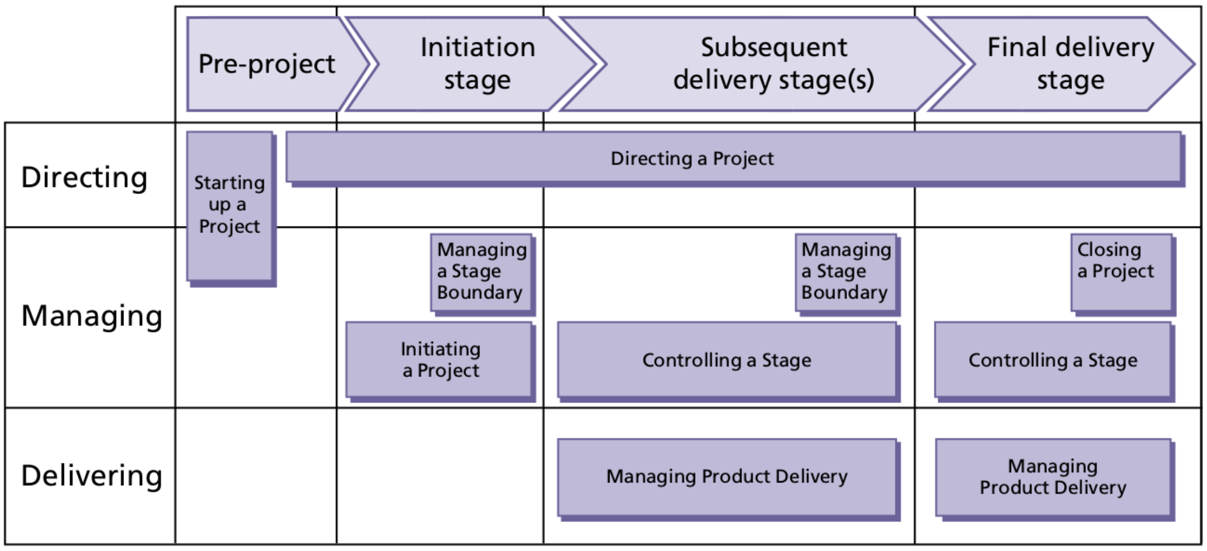

Kaufmännisches Projektmanagement
Messung und Analyse
Einführung und Pflege eines organisationsspezifischen Vorgehensmodells
Projektmanagement
V-Modell-Kern
Konfigurations-management
Qualitätssicherung
Problem- und Änderungsmanagement
V-Modell-Kern
V-Modell-Kern
Logistikkonzeption
Weiterentwicklung und Migration von Altsystemen
Systemerstellung
Benutzbarkeit und Ergonomie
Evaluierung von Fertigprodukten
Anforderungsfestlegung
HW-Entwicklung
Multi-Projektmanagement
SW-Entwicklung
Lieferung und Abnahme (AG)
Vertragsschluss (AG)
Lieferung und Abnahme (AN)
Vertragsschluss (AN)
Sicherheit
Sicherheit (AN)
Die Welt
Legende:
Alle V-Modell-Projekte
Organisationsspezifisches Vorgehensmodell
AG/AN-Schnittstelle
Systementwicklung
A B Um B auswählen zu können muss auch A gewählt werden
A B C C benötigt mindestens einen der Bausteine A bzw. B

Figure 5.2: Project Modules

5.6 PRINCE2

PRINCE2 (PRojects IN Controlled Environments) is a structured project management method and practitioner certification programme developed by the UK government.

- Seven Themes: Business Case, Organisation, Quality, Plans, Risk, Change, Progress
- Seven Principles: Continous Business Justification, Learn from Experience, Defined Roles and Responsibilities, Manage by Stages, Manage by Exception, Focus on Products, Tailor to Suit Project Environment
- Seven Processes: Directing, Starting up, Initiating, Managing Stage Boundaries, Controlling Stages, Managing Product Deliveries, Closing



www.axelos.com/best-practice-solutions/prince2/what-is-prince2

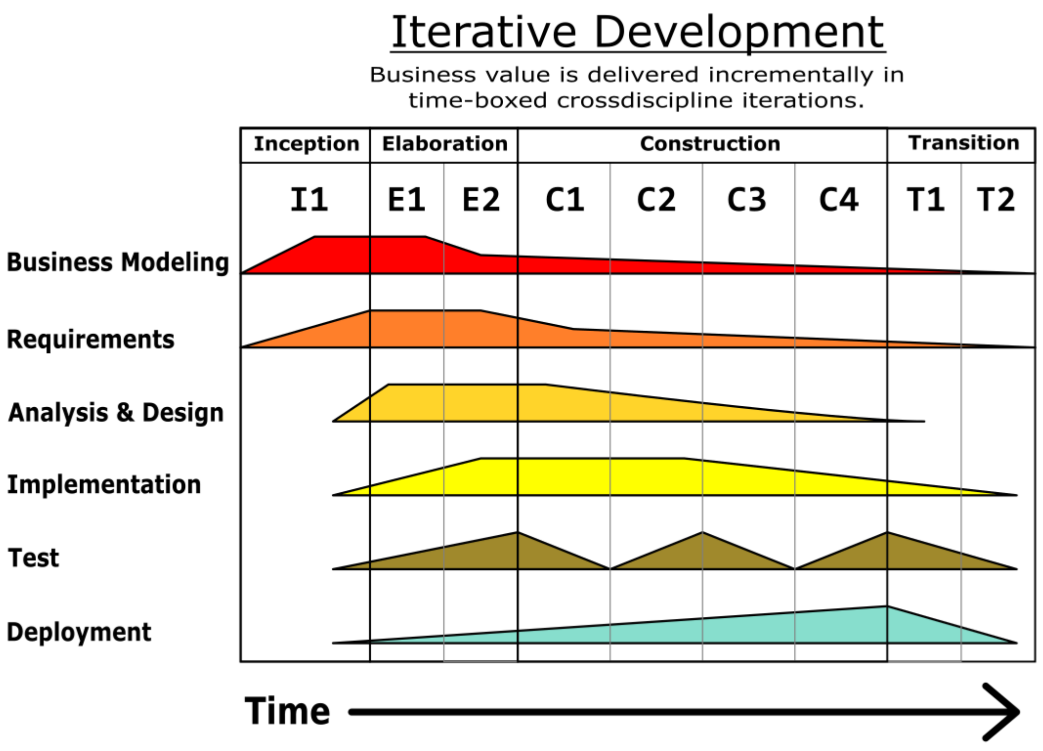
1. **Directing a Project (DP)** : Directing a Project runs from the start-up of the project until its closure. This process is aimed at the Project Board. The Project Board manages by exception, monitors via reports and controls through a number of decision points.
2. **Starting up a Project (SU)** : This is the first process in PRINCE. It is a pre-project process, designed to ensure that the pre-requisites for initiating the project are in place. The process expects the existence of a Project Mandate which defines in high level terms the reason for the project and what outcome is sought. Starting up a Project should be very short.
3. **Initiating a Project (IP)** : Agree whether or not there is sufficient justification and resources to proceed with the project. Document and confirm that an acceptable Business Case exists for the project and ensure that the investment of time and effort required by the project is made wisely, taking account of the risks to the project.
4. **Managing Stage Boundaries (SB)** : This process provides the Project Board with key decision points on whether to continue with the project or not.
5. **Controlling a Stage (CS)** : This process describes the monitoring and control activities of the Project Manager involved in ensuring that a stage stays on course and reacts to unexpected events. The process forms the core of the Project Manager's effort on the project, being the process which handles day-to-day management of the project.

6. **Managing Product Delivery (MP)** : The objective of this process is to ensure that planned products are created and delivered.
7. **Closing a Project (CP)** : The purpose of this process is to execute a controlled close to the project. The process covers the Project Manager's work to wrap up the project either at its end or at premature close. Most of the work is to prepare input to the Project Board to obtain its confirmation that the project may close.

5.7 Unified-Process-Model

The Unified-Process-Model is

an iterative and incremental development process developed by Ivar Jacobson, Grady Booch and James Rumbaugh (1999).



en.wikipedia.org/wiki/Unified_Process

Characteristics

use-case driven: uses cases define what the system is expected to do for the different types of users.

iterative and incremental : based on a series of iterations each adding improved functionality.

architecture-centric: the fundamental structure of the system, its elements and their relations builds the foundation of the development.

risk-focused: critical risks are focused early in the life cycle.

5.7.1 Development Phases

Inception: Establish an approximate vision of the project, describe the business case, define the scope and make a rough estimate for the cost and schedule.

Elaboration: definition of requirements with use cases, risk factors identification, system architecture design

Construction: implementation of system features (use cases) in a series of short iterations with executable releases.

Transition: deployment of the system to the target users, training, process and data migration

5.7.2 Development Disciplines

Business modeling: Define business objectives: increase process speed, reduce cycle time, increase quality (customer satisfaction, availability, reaction time), reduce costs (maintenance, labor, materials, scrap, or capital costs)

Requirements: definition of major functions with use cases, attributes, capabilities, characteristics, that present values to a customer, organization, internal user, or other stakeholder.

Analysis & Design: analyse the requirements and build the system architecture by decomposing the system into components and their interfaces,

Implementation: programming and building the system with its components and interfaces,

Test: planning, specifying, designing, executing system tests

Deployment: delivery, installation and putting into service

5.7.3 Use Case

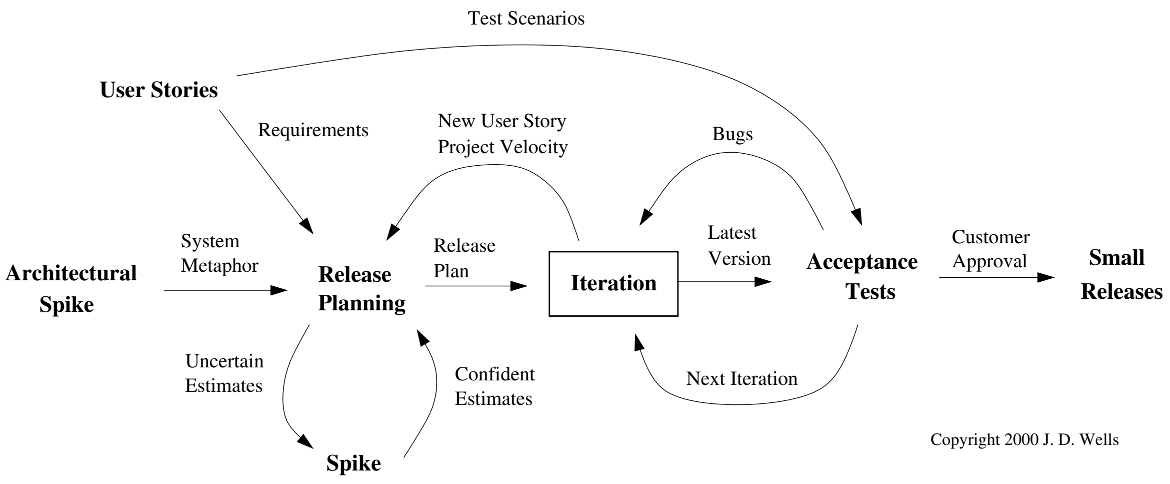
The sequence of actions in response to input of one or more actors:

Title	Name of use case
Description	some short text describing the scope.
Actor(s)	person(s) who interact with this particular use case.
Precondition	anything that this use case can assume to be true prior to beginning it's life cycle.
Success scenario	a sequence of steps describing correct flow of events that take place.
Extensions	flow of application when it deviates from success scenario's flow • Alternate flows - other options of correct flow • Exception flows - flow of events for when things go wrong
Post condition	state of application after everything is done

5.8 Extreme Programming (XP)

XP is a set of engineering practices developed by Kent Beck beginning 1990 and used at Chrysler 1996 in the C3 project with focus on software quality improvement and responsiveness to changing user needs. (www.extremeprogramming.org)

Based on the values communication, simplicity, feedback, respect, and courage.



Planning:

- User Stories are written.
- Release Planning creates the release schedule.
- Make frequent small releases
- The Project velocity is measured.
- The Project is divided into iterations.
- A stand up meeting starts each day.
- Move people araound.

Coding:

- The customer is always available.
- Code must be written to agreed standards.
- Code the unit test first.
- All production code is pair programmed.
- Integrate often.
- Correctness first, then optimize.
- 40-hour weeks, no overtime.

Design:

- Simplicity.
- Choose a system metaphor.
- Use CRC cards for design sessions.
- Create spike solutions to reduce risk.
- No functionality is added early.
- Refactor whenever and wherever possible.

Testing:

- All code must have unit tests.
- All code must pass all unit tests before it can be released.
- When a bug is found tests are created.
- Acceptance tests are run often and the score is published.

5.8.1 Comparing User Stories and Use Cases

User Stories and Use Cases both identify users and describe the values to be achieved. User Stories are more informal, Use Cases are focused on functional specifications.

A “User Story” is

- …a short, informal description of how some class of user could interact with and benefit from the proposed software (Kent Beck).
- …a simple, clear, brief descriptions of functionality that will be valuable to real users (Mike Cohn).
- …a very high-level definition of a requirement, containing just enough information so that the developers can produce a reasonable estimate of the effort to implement it (Scott W. Ambler).

Examples (www.agilemodeling.com/artifacts/userStory.htm):

- Students can purchase monthly parking passes online.
- Parking passes can be paid via credit cards.
- Professors can input student marks.
- Students can obtain their current seminar schedule.
- Students can order official transcripts.
- Students can only enroll in seminars for which they have prerequisites.
- Transcripts will be available online via a standard browser.

Mike Cohn suggests to create User Stories using the template

As a (role) I want (something) so that (benefit)

Example:

- As a student I want to purchase a parking pass so that I can drive to school.

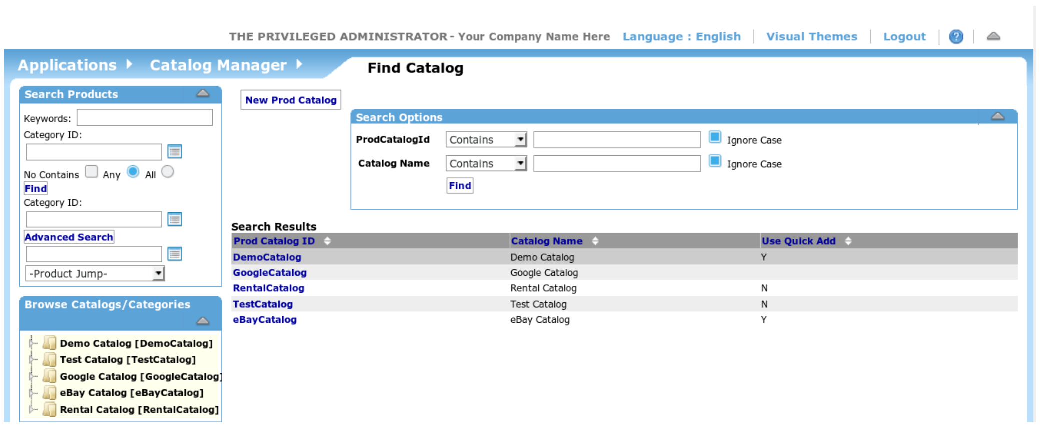
Use Case Example: Apache OFBIZ Project

OFBIZ is a suite of business applications flexible enough to be used across any industry. It includes the features Product & Catalog Management, Promotion & Pricing Management, Supply Chain Fulfillment, Contracts, Payments & Billing. ofbiz.apache.org

Title	Create/Edit Catalog
Description	User creating the catalog (associated with the product store) and updating its information.
Actor(s)	Catalog Manager
Precondition	The Catalog Manager accesses the catalog.
Success scenario	1. User selects ’Catalogs’ menu item. 2. System displays criteria for search and list of catalogs present in the system. 3. User clicks on “New Prod Catalog” button. 4. System displays a form to create the new catalog. 5. User enters the information and click on “Update” button. 6. System successfully creates a new catalog. 7. User again selects ’Catalogs’ menu item. 8. User Clicks on any catalog Id. 9. System displays edit product catalog page. 10. User enters/updates the information and click on “Update” button. 11. System successfully updates the catalog.
Post condition	User is able to Create/Edit catalog.

User Story:

Create/Build Category Hierarchy	**Priority: 1 Story Points: 5**
User creates new category/categories in the system. There can be multiple levels of category hierarchy in the system. User can associate/dissociate the next level (child) categories with the category. User can also update the name of the category, the changes will be reflected throughout ERP instantly. Apart from it, User associates product with the category and user can also copy the products to another category.	



Demo Catalog [DemoCatalog]
Test Catalog [TestCatalog]
Google Catalog [GoogleCatalog]
eBay Catalog [eBayCatalog]
Rental Catalog [RentalCatalog]

Search Results

Prod Catalog ID	Catalog Name	Use Quick Add
DemoCatalog	Demo Catalog	Y
GoogleCatalog	Google Catalog	
RentalCatalog	Rental Catalog	N
TestCatalog	Test Catalog	N
eBayCatalog	eBay Catalog	Y

Figure 5.3: demo-stable.ofbiz.apache.org/catalog/control/FindCatalog

5.8.2 INVEST (Bill Wake)

I	Independent	The user story should be self-contained, in a way that there is no inherent dependency on another user story.
N	Negotiable	User stories, up until they are part of an iteration, can always be changed and rewritten.
V	Valuable	A user story must deliver value to the end user.
E	Estimable	You must always be able to estimate the size of a user story.
S	Small	User stories should not be so big as to become impossible to plan/task/prioritize with a certain level of certainty.
T	Testable	The user story or its related description must provide the necessary information to make test development possible.

5.9 Scrum

Scrum[1] is a lightweight, iterative and incremental management framework. Originally developed by Hirotaka Takeuchi and Ikujiro Nonaka in 1986 for product development Scrum has been adapted to software developmment by Jeff Sutherland and Ken Schwaber (1996).

The scrum team is a cross-functional, self-managing group of people who have all the skills necessary to create value and decide internally who does what, when and how:

- **Developers**: do the work to build increments during a sprint,
- **Scrum Master**: coaches and supports the team members, removes impediments,
- **Product Owner**: represents the product stakeholders, defines the product backlog and goal.

- The Scrum process is divided into **sprints** each with a duration of 1 to 4 weeks.
- The work to be done is defined by the **product backlog**, which contains a collection of user stories, bug fixes and non-functional requirements.
- Before starting a sprint the developers select a subset of items from the product backlog for the **sprint backlog** such that the agreed sprint goal can be reached within duration of the sprint.
- During the sprint no changes that would endanger the sprint goal are made in the sprint backlog. The product backlog is refined as needed and the scope may be clarified and renegotiated with the **product owner** as the understanding of the project is growing.

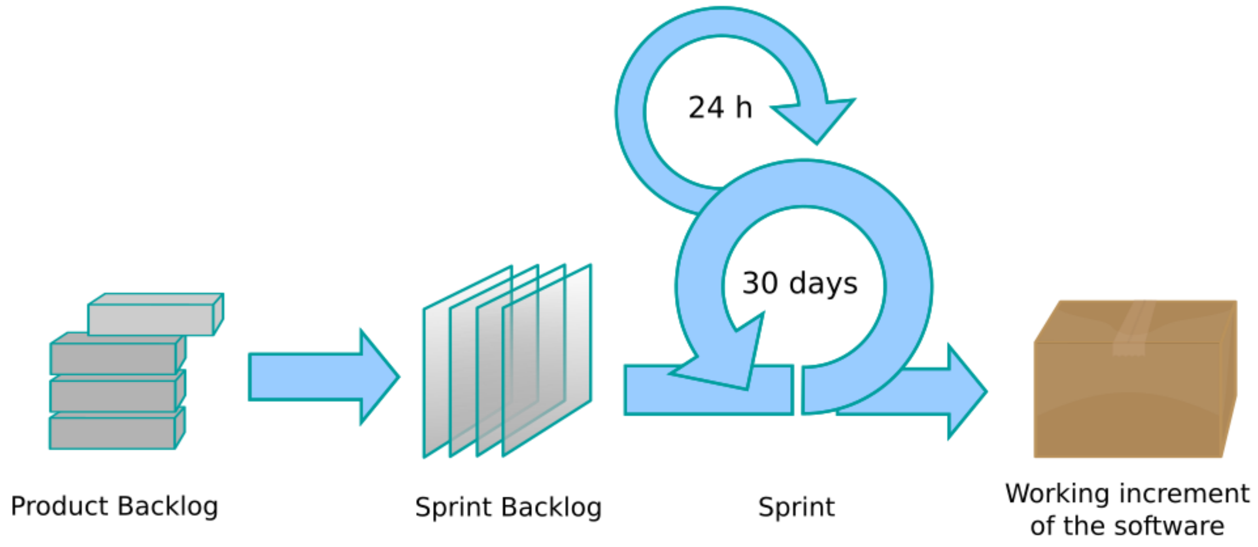


Figure 5.4: Scrum Process (Wikipedia)

[1]Rugby: a formation of players

Daily Scrum

On each day of the sprint, all team members attend a daily Scrum meeting (maximum duration 15 minutes) where each attendee answers the questions:

1. What did you do yesterday?
2. What will you do today?
3. Are there any impediments in your way?

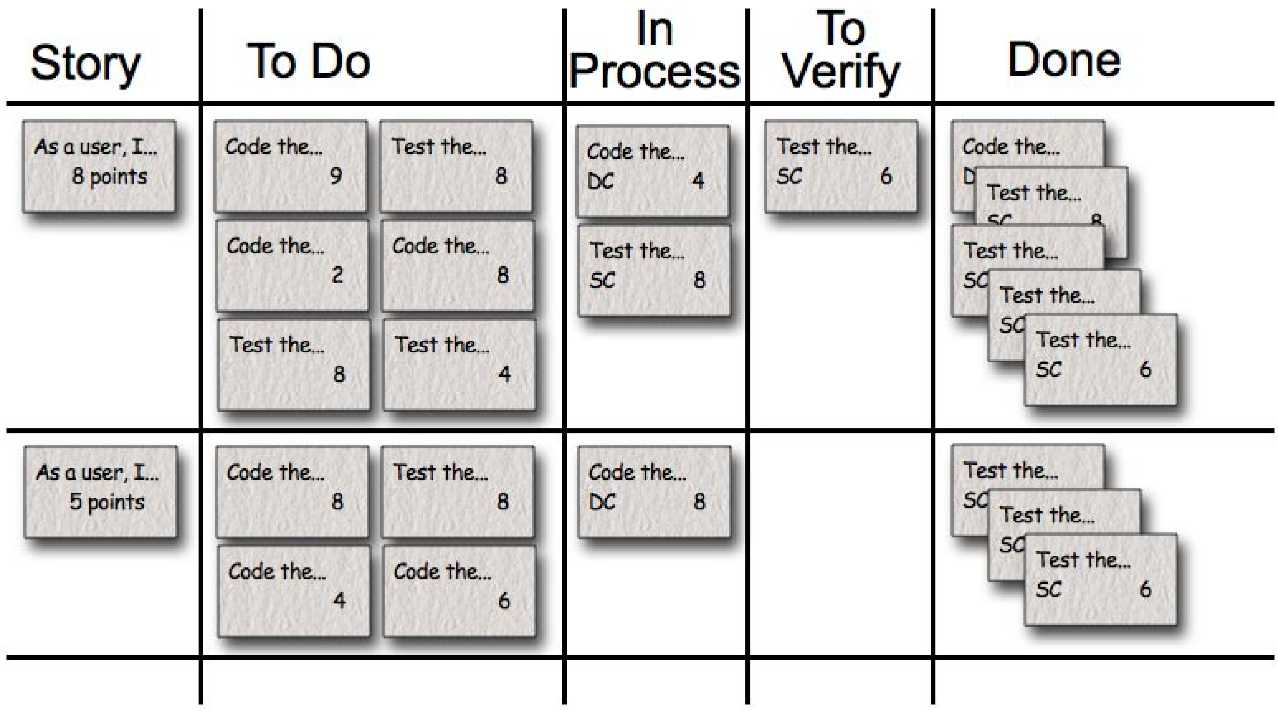


Figure 5.5: A Scrum Taskboard (www.mountaingoatsoftware.com)

5.10 Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

agilemanifesto.org

5.10.1 Manifesto for Software Craftmanship

As aspiring Software Craftsmen we are raising the bar of professional software development by practicing it and helping others learn the craft. Through this work we have come to value:

Not only working software but also well-crafted software

Not only responding to change but also steadily adding value

Not only individuals and interactions but also a community of professionals

Not only customer collaboration but also productive partnerships

That is, in pursuit of the items on the left we have found the items on the right to be indispensable.

manifesto.softwarecraftsmanship.org

Kent Beck (Extreme Programming explained)

> Listening, Testing, Coding, Designing. That's all there is to software. Anyone who tells you different is selling something.

5.11 Exercises

1. Discuss with your team members which development model fits best to implement the Gene Information Service.
2. Discuss the pros and cons between the waterfall model and an agile approach.

6 Requirements Engineering

ISO/IEC/IEEE 29148-2011 *Systems and Software Engineering - Life Cycle Processes - Requirements Engineering*

> ...is an **interdisciplinary** process concerned with discovering, eliciting, developing, analyzing, determining verification methods, validating, communicating, documenting, and managing requirements.

Glossary

Requirement: statement which translates or expresses a need and its associated constraints and conditions to solve a problem or to achieve an objective.

Stakeholder: individual or organization having a right, share, claim, or interest in a system or in its possession of characteristics that meet their needs and expectations. Stakeholders include, but are not limited to, end users, end user organizations, supporters, developers, producers, trainers, maintainers, disposers, acquirers, customers, operators, supplier organizations, accreditors, and regulatory bodies.

6.1 Process

Elicitation: build an understanding of the problem that the software is required to solve. It is fundamentally a human activity and is where the stakeholders are identified and relationships established between the development team and the customer.

Analysis: detect and resolve conflicts between requirements, discover the bounds of the software and how it must interact with its organizational and operational environment, elaborate system requirements to derive software requirements.

Architectural Design: the system structure with its components, their connections and interfaces.

Specification: create a document that can be systematically reviewed, evaluated, and approved.

- Stakeholder Requirements Specification
- System Requirements Specification
- Software Requirements Specification

Validation: verify that a requirements document conforms to company standards and that it is understandable, consistent, and complete

Questions

- **Who** is involved? (User profiles, Stakeholders, Responsibilities, Organization)
- **What** is the current state? (Business processes, problems, weaknesses, oportunities)
- **When** must the system be operational? (Timeline, milestones)
- **Where** will the system be installed? (Hardware and Software environment)
- **Why** is it necessary to develop a new system? (goals, benefits, risks)
- **What** functions and performance should the system provide? (use cases, user stories, data size, concurrent access)
- **Which** restrictions have to be considered? (programming languages, availability, response time, storage etc.)

Outcome

- Software Requirements Specification (SRS)

  A SRS document is crafted at the initial phase of a project (to kick off the project). Depending on the used development model, requirements will be converted into

  - User Stories (Agile Approach)
  - Use Cases (Unified Process)
- Project plan

Sources of Information:

- Interviews, Workshops, Scenarios, Observations
- Literature: Books, Journals, Manuals
- Documents, Forms, Instructions, Regulations
- Internet

6.2 Functional Requirements

Functional requirements describe the functions that the software is to execute.

- Inputs (Events, Triggers)
- Outputs (responses, results)
- Preconditions (additional information needed)
- Actions
- Postconditions
- Side effects

6.3 Nonfunctional Requirements

Nonfunctional requirements (NFR) describe restrictions and limitations that need to be considered for the system development:

- usability, error handling
- documentation (installing, maintaining, using)
- portability, compatibility (hardware, operating systems, software interfaces)
- performance, scalability (response times, concurrent users, workload)
- safety, security (access permissions, protection against damage, attacks)
- reliability, availability, maintainability

Examples: en.wikipedia.org/wiki/Non-functional_requirement

6.4 Software Requirements Specification (SRS)

The Software Requirements Specification (SRS) establishes the basis for agreement between customers and contractors or suppliers (in market-driven projects, these roles may be played by the marketing and development divisions) on what the software product is to do as well as what it is not expected to do. It should also provide a realistic basis for estimating product costs, risks, and schedules.

Organizations can also use a SRS document as the basis for developing effective verification and validation plans.

Software Requirements Specification SRS

IEEE Std 830, ISO/IEC/IEEE 29148

Title
What, Who, When, Where
Content
Summary

1. Introduction
1.1 Purpose
of this SRS and intended audience
1.2 Scope
summary of core functions, benefits, objectives
1.3 Definitions and Acronyms
1.4 References
1.5 Overview

2. Overall Description
2.1 Product Perspective
business case and context, architecture, deployment diagram
2.2 Product Functions
major functional capabilities, use case diagram
2.3 User Characteristics
user classes (actors), skills and knowledge
2.4 General Constraints
standards, rules, regulations, target platforms
2.5 Assumptions and Dependencies
factors that impact the requirements

3. Specific Requirements
Definition of functional and nonfunctional requirements using:
- Use Cases
- GUI mockups
- UML diagrams (classes, sequences, state-transitions ...)
- data flow diagrams, data catalogs
- Entity relationship diagrams (database model)
- file formats, communication protocols

6.4.1 Exercise

Search, select and evaluate a SRS document using the following criteria:

- complete
- correct
- consistent
- verifiable
- unambigous
- traceable
- prioritised
- feasible

7 Software Version Control

Version control (also known as revision control, source control, or source code management) is a class of systems responsible for managing changes to computer programs, documents, web sites, or other collections of information. Version control is a component of software configuration management.

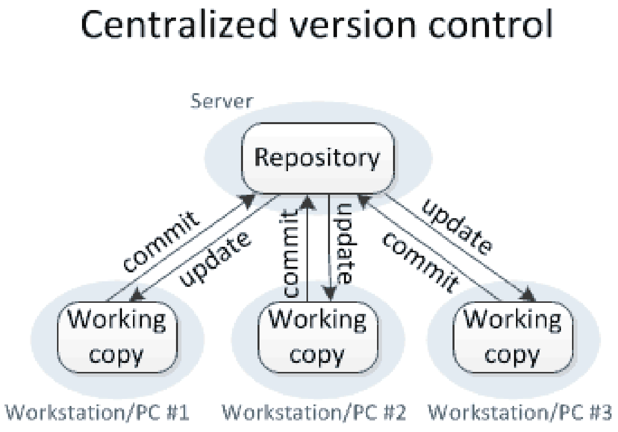
A version control system serves the following purposes, among others:

1. Version control gives access to historical versions of your project.
2. You can reproduce and understand a bug report on a past version of your software.
3. Version control enables multiple people to simultaneously work on a single project. Each person edits his or her own copy of the files and chooses when to share those changes with the rest of the team.

There are a number of available systems for version control. They all could be divided into two categories:

- Centralized

  In centralized version control, each user gets his or her own working copy, but there is just one central repository. As soon as you commit, it is possible for your co-workers to update and to

  see your changes.

  For others to see your changes, 2 things must happen:

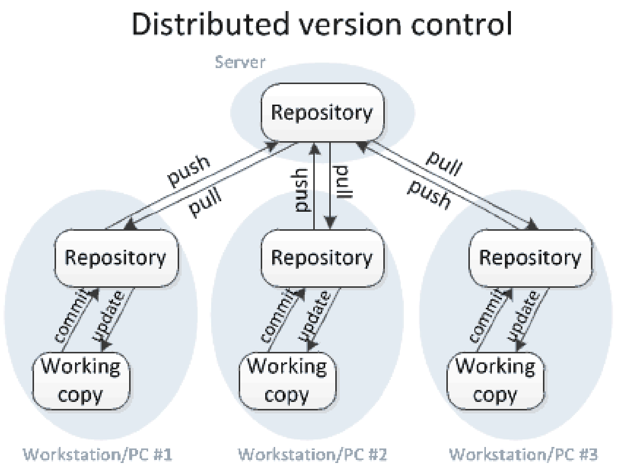
  - You commit
  - They update

Typical systems are:

- CVS
- Subversion

- Distributed

  In distributed version control, each user gets his or her own repository and working copy. After you commit, others have no access to your changes until you push your changes to the central repository. A pull needs to be executed to get new data into your local repository.

  For others to see your changes, 3 things must happen:

  - You commit
  - You push
  - They pull (fetch)

Typical systems are:

- GNU Arch
- Bazaar-NG
- BitKeeper
- Git
- Monotone
- Mercurial

There is also another way to classify version control systems:

Lock-Modify-Unlock (Pessimistic Revision Control) Many version control systems use a lock-modify-unlock model to address the problem of many authors clobbering each other's work. In this model, the repository allows only one person to change a file at a time. This exclusivity policy is managed using locks.

Copy-Modify-Merge (Optimistic Revision Control) Other version control systems use a copy-modify-merge model as an alternative to locking. In this model, each user's client contacts the project repository and creates a personal working copy. This is a local reflection of the repository's files and directories. Users then work simultaneously and independently, modifying their private copies. Finally, the private copies are merged together into a new, final version. The version control system often assists with the merging, but ultimately a human being is responsible for making it happen correctly.

7.1 Git

Git is software for tracking changes in any set of files, usually used for coordinating work among programmers collaboratively developing source code during software development. Its goals include speed, data integrity, and support for distributed, non-linear workflows (several branches running on different systems).

Git was created by Linus Torvalds in 2005 for development of the Linux kernel, with other kernel developers contributing to its initial development. Since 2005, Junio Hamano has been the core maintainer. As with most other distributed version control systems, and unlike most client–server systems, every Git directory on every computer is a full-fledged repository with complete history and full version-tracking abilities, independent of network access or a central server. Git is free and open-source software

The most important features are:

- **Distributed repositories**: each user gets his or her own repository and working copy.
- **Branches and tags**: Branches and tags are part of the repository (compared to centralized systems)
- **Revision identifier**: The SHA1 of the commit is the hash of all the information.
- **Commits are snapshots**: In other version control systems, changes to individual files are tracked and referred to as revisions, but with git you are tracking the entire workspace, so they use the term snapshot to denote the difference.
- **Stages**: Files have one of the following stages: untracked, modified, staged, committed

7.1.1 Git Object Store

The object store contains the original data files and all the log messages, author information, dates, and other information required to rebuild any revision or branch of the project. There are 4 different object types:

- **Blob**: A blob is used to store file data. It is generally a file.
- **Tree**: A tree is basically like a directory. It references a bunch of other trees and blobs

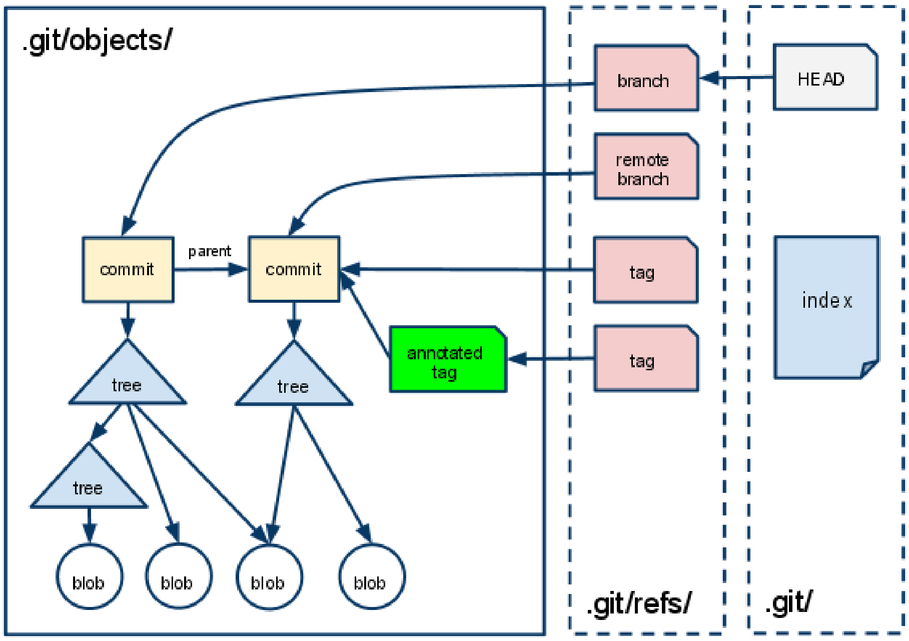
- **Commit**: A commit object holds metadata for each change introduced in the repository, including the author, committer, commit-data, and log messages.
- **Tag**: A tag object assigns an arbitrary human-readable name to a specific object usually a commit.

All Git objects have a 160 bit hash key:

`6ff87c4664981e4397625791c8ea3bbb5f2279a3`

There is a very high probability, that Git objects are globally unique.

Git objects are stored in the `.git` directory. They are reflecting a directed acyclic graph:



One of the biggest advantages of Git compared to other systems is the easy handling of branches. Branches are needed for:

- executing experiments
- try out new ideas
- development of new features

Or in general, for any type of modification. In some software projects, it is forbidden to push directly to the master branch. Therefore it is needed to create a branch for each modification.

7.1.2 Git commands

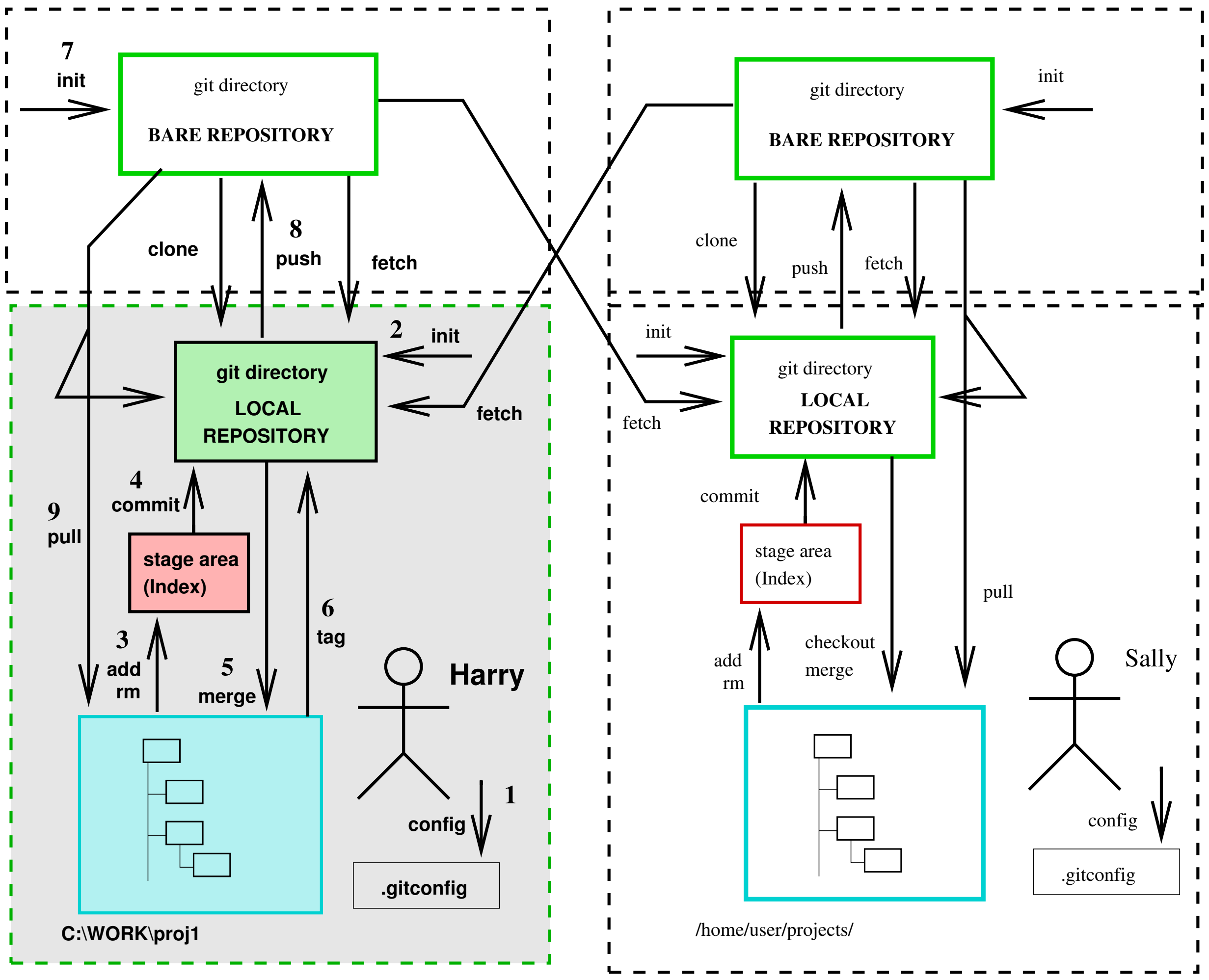


Figure 7.1: Workflow with git

1. config: Define user name and mail address:

   ```
   $ git config —global user.name "David Herzig"
   $ git config —global user.email "dave.herzig@gmail.com"
   ```

   This information will be stored in the home directory in a file called `.gitconfig`.

   There are 3 locations/levels for config files:

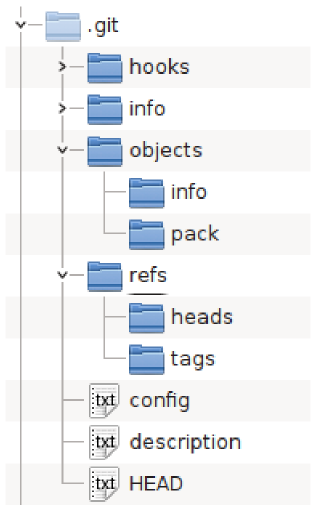
   - User level
   - Repository level
   - System level

   If you remove the `--global` switch, the configuration will be stored in the current repository.

2. init: create a new repository.

   Create an empty local repository in the current directory:

```
$ cd /home/user/projects
$ git init project1
Initialized empty Git repository in /home/user/projects/project1/.git
```



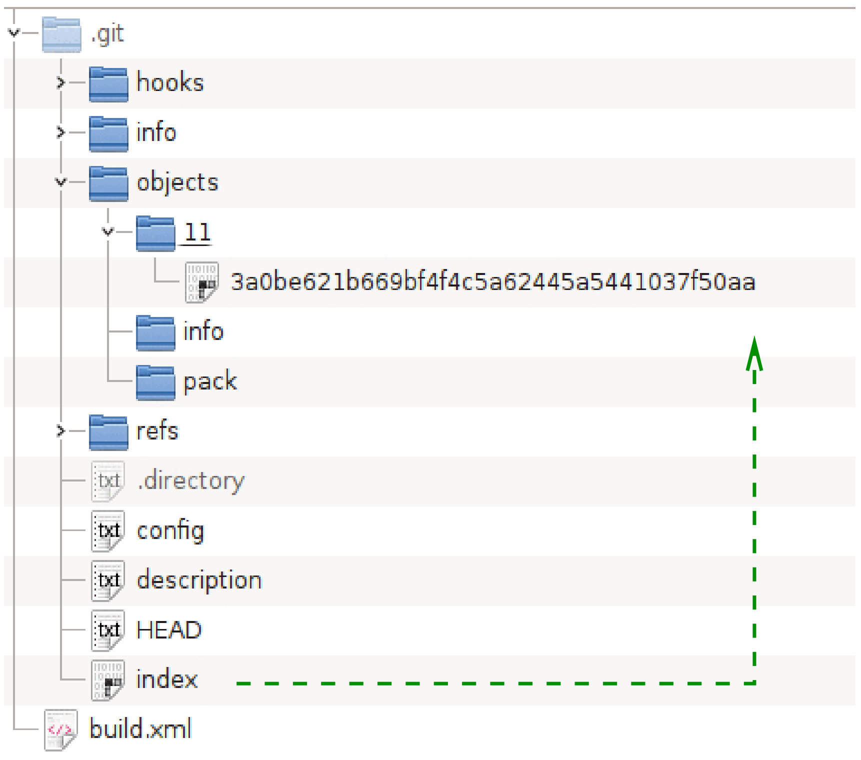
Note: To create a local repository from an existing one, the command `init` needs to be replaced with `clone`.

```
$ git clone file:///var/git/project1.git
```

3. add: Adds files in the to the staging area for Git.

```
$ cd project1
$ git add .
```

All files in the current directory and sub directories will be added.

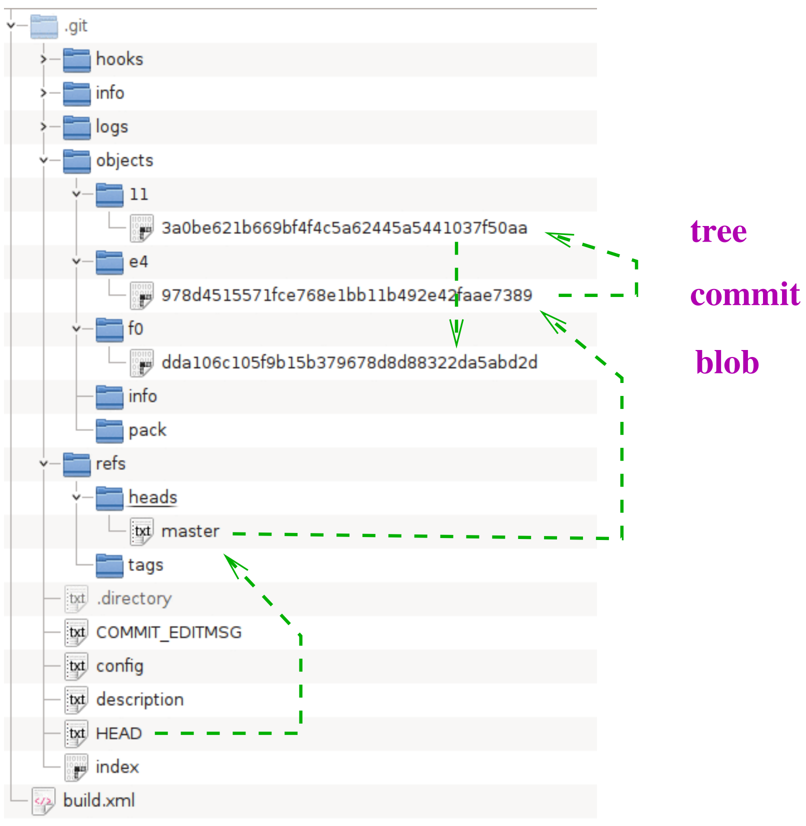


4. commit: Record the changes made to the files to a local repository. For easy reference, each commit has a unique ID:

   ```
   $ git commit -m "initial commit"
   ```

   Note: modified files needs to be added with the `add` command. Use the option `-u` to add all modified files.

   ```
   $ git commit -am "another commit"
   ```



```
.git
  hooks
  info
  logs
  objects
    11
      3a0be621b669bf4f4c5a62445a5441037f50aa
    e4
      978d4515571fce768e1bb11b492e42faae7389
    f0
      dda106c105f9b15b379678d8d88322da5abd2d
    info
    pack
  refs
    heads
      master
    tags
  .directory
  COMMIT_EDITMSG
  config
  description
  HEAD
  index
build.xml
```

5. checkout: To start working in a different branch, use git checkout to switch branches.

   ```
   $ git checkout -b ticket-53
   ```

6. tag: Git has the ability to tag specific points in a repository's history as being important. Typically, people use this functionality to mark release points:

   ```
   $ git tag -a rel-0.0 -m "Release 0.0"
   ```

   Note: Git supports two types of tags: lightweight and annotated. A lightweight tag is very much like a branch that doesn't change. It's just a pointer to a specific commit. Annotated tags, however, are stored as full objects in the Git database. They're checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG). It's generally recommended that you create annotated tags so you can have all this information; but if you want a temporary tag or for some reason don't want to keep the other information, lightweight tags are available too.

   Display all tags:

   ```
   $ git tag -l
   ```

7. init: This command turns a directory into an empty Git repository. This is the first step in creating a repository. After running git init, adding and committing files/directories is possible.

   ```
   $ git init −−bare /var/git/project1.git
   ```

8. push: Sends local commits to the remote repository. git push requires two parameters: the remote repository and the branch that the push is for.

   ```
   $ git remote add origin file:///var/git/project1
   $ git push origin master
   ```

   Note:

   - User needs write permission on the repository.
   - With "remote add" an alias for a remote repository will be generated (in this case origin)
   - Display all remote repositories URL:

     ```
     $ git remote −v
     ```

   - Tags are not automatically transferred to the remote repository. This needs to be done in addtion:

     ```
     $ git push origin rel−0.0
     ```

     or all together:

     ```
     $ git push origin −−tags
     ```

9. pull: To get the latest version of a repository run git pull. This pulls the changes from the remote repository to the local computer.

   ```
   $ git pull
   ```

Note: the default repository and default branch could/should be defined in the Git configuration file.

```
git config branch.master.remote origin
git config branch.master.merge refs/heads/master
```

These values are automatically set after a `git clone` operation.

7.1.3 Additional operations

- Get a specific version:

  ```
  $ git checkout −b branch−0 rel−0.0
  ```

- Display changes of a file:

  ```
  $ git log Main.java
  ```

- Detailed display of the last 2 changes of a file:

  ```
  $ git log −p −2 Main.java
  ```

- Detailed display of the changes in the last 2 weeks:

  ```
  $ git log −−sinde=2.weeks
  ```

- Display changes of a file in the stage area:

  ```
  $ git diff Main.java
  ```

- Compare stage area and repository:

  ```
  $ git diff −−staged
  ```

- Remove files:

  ```
  $ git rm Main.java
  ```

  The file will be marked as deleted after the commit.

- rename a file:

  ```
  $ git mv Main.java NewMain.java
  ```

  The changes will transferred into the repository after the commit.

- display current state

  ```
  $ git status
  ```

7.1.4 Branches

A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is master. As you start making commits, you're given a master branch that points to the last commit you made. Every time you commit, the master branch pointer moves forward automatically. Changes are normally done on a branch. After the change is completed, the branch will be merged back into the master branch.

- Create new branch:

  ```
  $ git checkout −b ticket−53
  ```

- Display all branches:

  ```
  $ git branch
  ```

  Note: remote branches will be displayed by using the option `-a`

- Commit changes:

  ```
  $ git commit −m ”initial commit”
  ```

- Change and merge branch

  ```
  $ git checkout master
  $ git merge ticket−53
  ```

  If there are no conflicts, the changes will be transferred into the repository.

- Delete branch

  ```
  $ git branch −d ticket−53
  ```

7.1.5 Conflicts

A conflict could occur if 2 or more developers work on the same file(s).

Example:

```
$ git push origin master
To file:///var/git/project1.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'file:///var/git/project1.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again.  See the
'Note about fast-forwards' section of 'git push --help' for details.
```

After a `git pull` the following messages could be displayed:

```
Auto-merging <filename>
CONFLICT (content): Merge conflict in <filename>
Automatic merge failed; fix conflicts and then commit the result
```

The lines with conflicts will be marked in the file:

```
<<<<<<<
=======
>>>>>>>
```

These conflicts needs to be fixed manually.

7.2 Exercise

1. Perform the following steps on the data loader component:
   - Create a repository (e.g. Github, Gitlab)
   - Push your code into that repository
   - Think about the following questions:
     Which parts of the project should be in the repository? Which parts of the project should NOT be in the repository?
   - Create a tag which marks the software with `V0_1`
   - Verify in the web console (Github, Gitlab) if everything is correct.
   - Create a branch and perform a modification on that branch.
   - Merge the branch into the master branch.

7.3 Software and further Informations

- Git project: git-scm.com
- Git tutorials: www.gitguys.com, www.atlassian.com/git/tutorials
- 8 ways to share your git repository (Patrick Debois): http://agile.dzone.com/news/8-ways-share-your-git
- Git Workflows www.atlassian.com/git/workflows
- Pro Git (Benutzeranleitung) progit.org/book
- Git Reference: gitref.org
- Git – Verteilte Versionsverwaltung für Code und Dokumente, (V. Haenel, J. Plenz, Open Source Press, 2011) gitbu.ch
- Eclipse-Plugin EGit: www.eclipse.org/egit
- TortoiseGit: code.google.com/p/tortoisegit
- Configuration management with Subversion, Ant and Maven (Gunther Popp) dpunkt.verlag, 2008
- Additional versioning tools:
  - Mercurial: mercurial.selenic.com
  - Monotone: www.monotone.ca
  - GNU arch: www.gnu.org/s/gnu-arch

8 The Software Build Process

Build is the process of creating the application program for a software release, by taking all the relevant source code files and compiling them and then creating a build artefact, such as binaries or executable program, etc.

The build process should be automated as much as possible and be reproducible.

You can also say that the build process is a combination of several activities which varies for each programming language and for each operating system but please remember the basic concepts are universal.

The build process may include the following activities:

- Fetching the code from source control repository
- Compile the code and check dependencies
- Run automated unit tests
- Link the libraries, code etc accordingly
- Build artifacts
- Deploy application
- Generate documentation
- Archive build logs

To have the possibility to create an automated and reproducible process, a description of each software (e.g. compiler) including parameters and dependencies is needed.

The build process is **CRISP** oriented.

Complete **R**epeatable **I**nformative **S**chedulable **P**ortable

Typical tools which could be used are::

- make, CMake and Automake/Autotools (Unix, Windows/Cygwin)
- Apache Ant
- Apache Maven
- Gradle

8.1 Apache Ant

Apache Ant is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. The main known usage of Ant is the build of Java applications. Ant supplies a number of built-in tasks allowing to compile, assemble, test and run Java applications. Ant can also be used effectively to build non Java applications, for instance C or C++ applications. More generally, Ant can be used to pilot any type of process which can be described in terms of targets and tasks.

To use Apache Ant, a build file in XML format is needed. This file has the following properties:

- Each build file contains one project
- Each project contains one or more **Targets**. Each Target will execute a given number of **Tasks**.

Sample build file:

```
<project name="SimpleProject" default="compile">

  <target name="init">
    <mkdir dir="build"/>
  </target>

  <target name="compile" depends="init">
    <javac srcdir="src"
         destdir="build"/>
  </target>
</project>
```

This build files takes care that

1. the directory `build` will be created (if not already there).
2. all Java source code files which are in the `src` directory will be compiled and the result will be written to the `build` directory.

A process based on Ant could be started in the following way:

Call	Description
ant	will execute the default target in the build file (build.xml)
ant -f otherbuild.xml	using a different name for the build file: otherbuild.xml
ant compile	executes the target **compile** aus.
ant -projecthelp	displays all available targets
ant -version	displays the current Ant version
ant -emacs	creates Emacs based log statements.

Per default, Ant contains about 80 tasks. Most of the work which needs to be done in the build process could be covered with the default tasks. Default tasks are:

- Copy
- Create directory
- Create jar file
- Compile code
- Execute operating system command

There is also the possibility to extend Ant with your own tasks. Many extension projects are available.

8.1.1 Constants

Whenever possible, make use of properties in your build file:

```
<property name="build.dir" value="build"/>
<target name="compile">
  <javac srcdir="src"
         destdir="${build.dir}"/>
</target>
```

Properties could be defined in an additional file or in the build file directly:

```
<property file="project.properties"/>
```

The project.properties file contains key value pairs:

```
build.dir=build
```

8.1.2 Create Jar file

A jar file could be created with the jar task:

```
<target name="dist" depends="compile"
   description="generate the distribution" >
   <mkdir dir="dist" />
   <jar destfile="dist/project.jar"
        basedir="${build.dir}"/>
</target>
```

The attribute destfile defines the final file name and the basedir defines, which files should be included.

To create an executable jar file, the following manifest needs to be added:

```
  Main-Class: classname
```

classname is the class, which contains the `main` method. There is a possibility to do that directly in the Ant task:

```
   <jar destfile="dist/project.jar"
        basedir="${build.dir}">
      <manifest>
        <attribute name="Main-Class"
                   value="example.Main"/>
      </manifest>
   </jar>
```

8.1.3 Fileset

Filesets are used as a common way to bundle files:

```
<fileset dir="src">
  <include name="**/*.java"/>
  <exclude name="**/*Test*"/>
</fileset>
```

A group will be created, containing all files from the `src` directory ending with `.java` or containing the word `Test` in the filename.

8.1.4 Path

A path is a collection of resources (files). A path has an id and could then be used within another task:

```
<path id="compile.classpath">
  <fileset dir="lib">
    <include name="**/*.jar"/>
  </fileset>
</path>

<path id="run.classpath">
  <pathelement location="${build.dir}"/>
  <path refid="compile.classpath"/>
</path>
```

This approach is used to set the classpath in the compilation process:

```
<javac ...>
  <classpath refid="compile.classpath"/>
</javac>
```

8.1.5 Exercise

1. Create an Ant build file for the loader project. This build file should contain the following targets:
   - clean (delete all generated artifacts)
   - compile (compiles the java code)
   - dist (creates an executable jar file, containing the build date)

8.1.6 Software and further Informations

- Java Development with Ant (Erik Hatcher, Steve Loughan), Manning Publications
- The Apache ANT Project: ant.apache.org
- Apache Ant: en.wikibooks.org/wiki/Programming:Apache_Ant
- Ant Wiki: wiki.apache.org/ant/FrontPage
- Introduction to Ant:www.exubero.com/ant/antintro-s5.html

8.2 Maven

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.
Maven's primary goal is to allow a developer to comprehend the complete state of a development effort in the shortest period of time. In order to attain this goal, Maven deals with several areas of concern:

- Making the build process easy
  While using Maven doesn't eliminate the need to know about the underlying mechanisms, Maven does shield developers from many details.
- Providing a uniform build system
  Maven builds a project using its project object model (POM) and a set of plugins. Once you familiarize yourself with one Maven project, you know how all Maven projects build. This saves time when navigating many projects.
- Providing quality project information
  Maven provides useful project information that is in part taken from your POM and in part generated from your project's sources. For example, Maven can provide:
  - Change log created directly from source control
  - Cross referenced sources
  - Mailing lists managed by the project
  - Dependencies used by the project
  - Unit test reports including coverage

  Third party code analysis products also provide Maven plugins that add their reports to the standard information given by Maven.
- Encouraging better development practices
  Maven aims to gather current principles for best practices development and make it easy to guide a project in that direction.
  For example, specification, execution, and reporting of unit tests are part of the normal build cycle using Maven. Current unit testing best practices were used as guidelines:
  - Keeping test source code in a separate, but parallel source tree
  - Using test case naming conventions to locate and execute tests
  - Having test cases setup their environment instead of customizing the build for test preparation

  Maven also assists in project workflow such as release and issue management.
  Maven also suggests some guidelines on how to layout your project's directory structure. Once you learn the layout, you can easily navigate other projects that use Maven.
  While takes an opinionated approach to project layout, some projects may not fit with this structure for historical reasons. While Maven is designed to be flexible to the needs of different projects, it cannot cater to every situation without compromising its objectives.

8.2.1 Project Object Model POM

The pom.xml file is the core of a project's configuration in Maven. It is a single configuration file that contains the majority of information required to build a project in just the way you want.
The most important information is:

- **General information**: Project information, Package ...
- **Dependencies**: Dependencies to external libraries,
- **Build**: information for the build process (plugins),
- **Properties**: project specific properties (e.g. Java version),
- **Profile**: Build variants (e.g. PROD, DEV, TST),
- **Repositories**: 3rd party repositories
- Reporting: Documentation

All information is stored in a file called `pom.xml`:

```
  <project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                     http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>example</groupId>
  <artifactId>myapp</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>

  <name>My First App</name>
  <url>http://maven.apache.org</url>

  <dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.7.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.7.1</version>
    <scope>test</scope>
  </dependency>
  </dependencies>
</project>
```

General information will be described in the following way:

- **groupId**: unique identifier of the project organization, e.g. ch.fhnw.mscmi
- **artifactId**: unique identifier of the outcome (product)
- **version**: version of the artifact
- **packaging**: package type (e.g. war, jar)

- **name**: project name for the documentation
- **url**: website of the project

As a dependency there is the `Junit` framework listed. Other dependencies could be added (e.g. Apache Commons).
A dependency could have a scope (test, compile, provided, runtime, system) to define the usage area.

8.2.2 Build-Lifecycle

The build process in Maven contains the following phases:

- **compile**: compile the source code
- **test**: run unit tests
- **package**: package compiled source code into the distributable format (jar, war, ...)
- **integration-test**: process and deploy the package if needed to run integration tests
- **install**: install the package to a local repository
- **deploy**: copy the package to the remote repository

There are additional goals available:

- **clean**: deletes all generated files
- **site**: creates the project documentation

For the full list of each lifecycle's phases, check out the Maven Reference.

To create a jar file, the following 8 phases are included:

	Phase	Plugin	Goal
1	process-resources	maven-resources-plugin	resources:resources
2	compile	maven-compiler-plugin	compiler:compile
3	process-test-resources	maven-resource-plugin	resources:testResources
4	test-compile	maven-compiler-plugin	compiler:testCompile
5	test	maven-surefire-plugin	surefire:test
6	package	maven-jar-plugin	jar:jar
7	install	maven-install-plugin	install:install
8	deploy	maven-deploy-plugin	deploy:deploy

8.2.3 How to use Maven?

A simple example will explain how Maven could be used:

1. Create project (based on a template):

   ```
   mvn archetype:generate
     -DarchetypeArtifactId=maven-archetype-quickstart
     -DarchetypeVersion=1.4
   ```

   Some information is needed to create the project. This information could be provided as parameters or in an interactive mode:

- **groupId**: namespace, organization name
- **artifactId**: project identifier
- **version**: version identifier
- **package**: default package

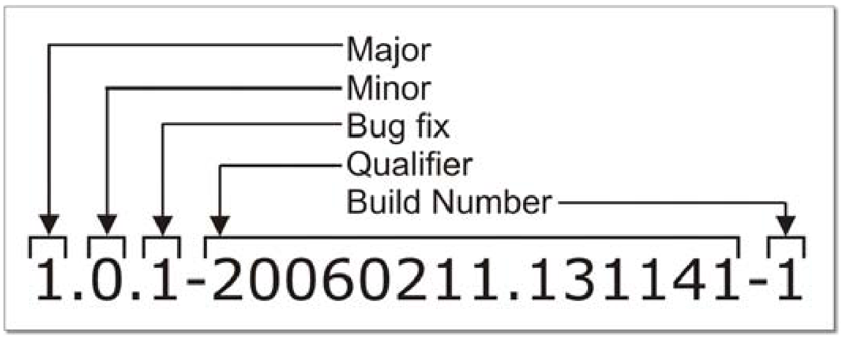


Figure 8.1: Format of the version identifier (Source: Better Builds with Maven)

Examples for version identifier:

- 1.0.1-20080211.131141-1
- 1.0-alpha
- 1.0

A project is now generated with the following file/directory structure:

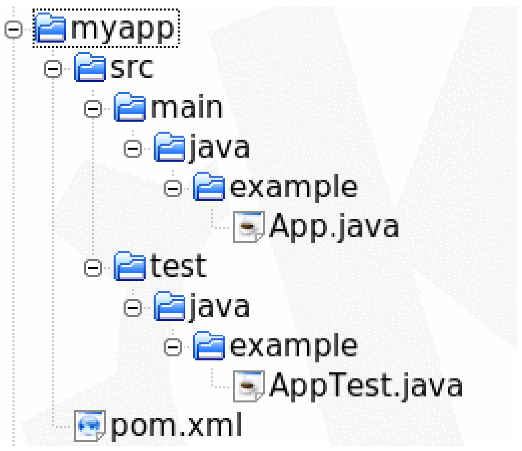


Figure 8.2: Directory structure of a Maven project

The available project templates could be displayed with the command `archetype:generate`. The option `-Dfilter` could be used search for specific templates.

2. Compile and Test A specific project phase could now be executed on command line:

```
mvn test
```

All phases from start to `test` will now be executed. That means:

- Dependencies will be solved
- Compile source code
- Execute unit tests

The Java version which is defined in the `JAVA_HOME` environment variable will be used to execute the maven build process.

Specific compiler flags will be defined in the `maven-compiler-plugin` plugin:

```
<plugins>
  <plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
      <source>1.8</source>
      <target>1.8</target>
    </configuration>
  </plugin>
</plugins>
```

3. Using 3rd/external party libraries Additional libraries will be added to the `dependencies` section:

```
<dependency>
    <groupId>org.biojava</groupId>
    <artifactId>core</artifactId>
    <version>1.9.5</version>
</dependency>
```

   Searching in the Maven repository is possible on the page: search.maven.org

   Maven will then download the libraries (if not already available) and stores the files into a local repository.

4. Running the program

   Executing a Java application does not belong to the core competencies of Maven. Nevertheless, there is a possibility to do that:

```
mvn exec:java -Dexec.mainClass=ch.fhnw.mscmi.App
```

   Additional arguments could be passed with `-Dexec.args=".."`.

   The exec plugin could also be configured

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <configuration>
    <mainClass>ch.fhnw.mscmi.App</mainClass>
  </configuration>
</plugin>
```

5. Package The package command takes the compiled code and package it in its distributable format, such as a JAR, WAR, EAR.
   Per default, the dependencies are not included. The assembly plugin could be used to achieve that:

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <configuration>
        <archive>
          <manifest>
            <mainClass>ch.fhnw.mscmi.App</mainClass>
          </manifest>
        </archive>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
    </plugin>
  </plugins>
</build>
```

With the goal **assembly:single** the application, including the dependencies, could be generated. This goal could also be combined with the package goal:

```
<executions>
  <execution>
    <id>make-assembly</id>
    <phase>package</phase>
    <goals>
      <goal>single</goal>
    </goals>
  </execution>
</executions>
```

Hiermit wird beim Aufruf von Maven mit dem Parameter **package** die Jar-Datei, welche alle Klassen der zu diesem Artefakt abhängigen Archiv-Dateien enthält.

6. Install Maven installs the created artefacts into the local repository. Therefore these artefacts are available for other projects:

```
mvn install
```

The path is build up in the following way:

`<groupId>/<artifactId>/<version>/<artifactId>-<version>.jar`

Example: org/example/myapp/1.0-SNAPSHOT/myapp-1.0.SNAPSHOT.jar

7. Deployment The deployment command will copy the artefacts into a remote repository. Several ways are supported: Copy, FTP, SCP/SSH:

```
<distributionManagement>
  <repository>
    <id>internal.repository</id>
    <name>Internal Repository</name>
    <url>file://${basedir}/target/deploy</url>
  </repository>
</distributionManagement>
```

For FTP and SCP/SSH: Credentials (username, password, public key) needs to be stored in the Maven settings file: `~/.m2/settings.xml`.

8.2.4 Properties

Like in Apache Ant, properties could be used to store values which are used at several positions in the pom.xml file:

```
  <dependencies>
    <dependency>
      ...
      <version>${junit.version}</version>
    </dependency>
  </dependencies>

<properties>
  <junit.version>5.7.1</junit.version>
</properties>
```

There are some implicit properties available in any Maven project these implicit properties are:

`project.*`	Maven Project Object Model (POM). You can use the `project.*` prefix to reference values in a Maven POM.
`settings.*`	You use the `settings.*` prefix to reference values from your Maven Settings in `~/.m2/settings.xml`.
`env.*`	Environment variables like `PATH` and `M2_HOME` can be referenced using the `env.*` prefix.
System Properties	Any property which can be retrieved from the `System.getProperty()` method can be referenced as a Maven property.

8.2.5 Build Profiles

With the help of profiles, several different builds could be defined (e.g. PROD, TST, DEV):

```
  <profiles>
    <profile>
      <id>production</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <build>
        <plugins>
          <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
              <debug>false</debug>
              <optimize>true</optimize>
            </configuration>
          </plugin>
        </plugins>
      </build>
    </profile>
  </profiles>
```

To run a specific profile, the option -P could be used when starting the Maven process:

```
  mvn -Pproduction compile
```

8.2.6 Filtern von Ressourcen

Für die einfache Anpassung umgebungsabhängiger Konstanten und Texte werden bei Java-Programmen in der Regel spezielle Ressourcen-Dateien verwendet. Zum Beispiel können damit der Name und die Version einer Applikation gesetzt werden:

```
ResourceBundle rb =
   ResourceBundle.getBundle("ApplicationResources");

String appname = rb.getString("application.name");
String version = rb.getString("application.version");
System.out.println("This is " + appname +
                   " (Version " + version + ")");
```

Legt man im Verzeichnis src/main/resources die Datei ApplicationResources.properties mit folgendem Inhalt ab

```
# application resources
application.name=${project.name}
application.version=${project.version}
```

dann kann Maven dafür sorgen, dass die Property-Werte in der process-resources-Phase aus der POM-Datei übernommen und eingesetzt werden.

Dazu muss jedoch das Resource-Element entsprechend konfiguriert werden:

```
 <build>
   <resources>
     <resource>
       <directory>src/main/resources</directory>
       <filtering>true</filtering>
     </resource>
   </resources>
</build>
```

8.2.7 Test-Durchführung

Die Durchführung der Unit-Tests ist Bestandteil des Standard-Build-Prozesses, für welches das Plugin maven-surefire-plugin zuständig ist:

```
mvn test
```

Ohne entsprechende Konfiguration werden alle Tests in den Dateien des Verzeichnisses src/test und aller Unterverzeichnisse mit folgendem Muster ausgeführt:

- `**/Test*.java`
- `**/*Test.java`
- `**/*TestCase.java`

Dabei können die Testklassen mit TestNG, Junit oder sogar als normale POJO-Klassen implementiert werden. Das Plugin berücksichtigt dazu die entsprechende Dependency-Konfiguration.

Die Ergebnisse werden als Report-Dateien jeweils in Text- und XML-Format ins Verzeichnis target/surefire-reports abgelegt und der Build-Prozess wird abgebrochen, falls ein Unit-Test fehlschlägt.

Dieses Verhalten kann jedoch konfiguriert werden:

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <includes>**/*</includes>
    <testFailureIgnore>false</testFailureIgnore>
  </configuration>
</plugin>
```

8.2.8 Reporting

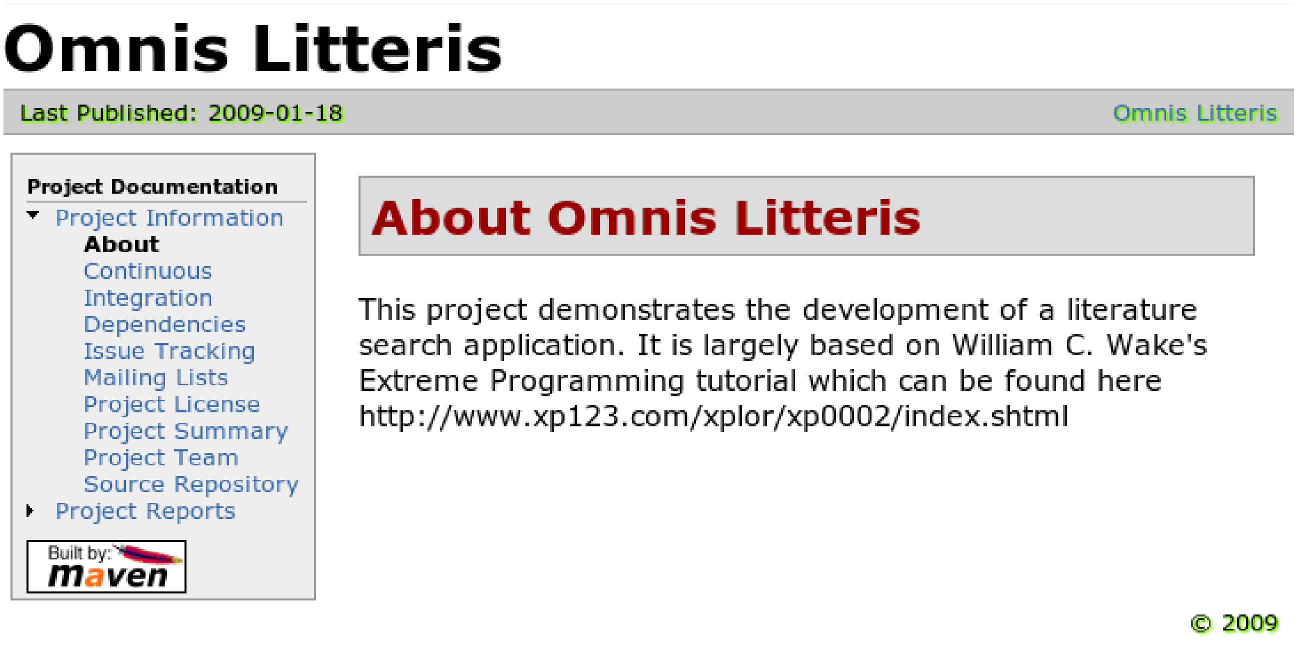
Mit der Anweisung:

```
% mvn site
```

erzeugt Maven eine Projekt-Dokumentation, die im Verzeichnis target/site abgelegt wird. Die dazu nötigen Informationen werden ebenfalls der POM-Datei entnommen. Zum Beispiel:

```
..
  <name>My first Project powered by maven</name>
   <url>http://your.project.url</url>
   <description>Write some brief explanatory text
                about your project.</description>
..
```

Das Layout der so erzeugten Dokumentation kann mit dem File src/site/site.xml angepasst werden.


Omnis Litteris

Last Published: 2009-01-18 Omnis Litteris

About Omnis Litteris

This project demonstrates the development of a literature search application. It is largely based on William C. Wake's Extreme Programming tutorial which can be found here http://www.xp123.com/xplor/xp0002/index.shtml

© 2009

Figure 8.3: Eine mit Maven erstellte Projektseite

Für eine umfassende Dokumentation sorgen zahlreiche Plugins (Beispiele):

- Change-Log: maven-changelog-plugin
- Checkstyle: maven-checkstyle-plugin

- Junit-Report: maven-surefire-report-plugin
- PMD: maven-pmd-plugin
- Findbugs: findbugs-maven-plugin
- JavaDoc: maven-javadoc-plugin
- Cobertura: cobertura-maven-plugin
- JXR (Cross-Reference): maven-jxr-plugin

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>cobertura-maven-plugin</artifactId>
    </plugin>
    <plugin>
      <artifactId>maven-javadoc-plugin</artifactId>
    </plugin>
    <plugin>
      <artifactId>maven-jxr-plugin</artifactId>
    </plugin>
..
  </plugins>
</reporting>
```

8.2.9 Eigene Dateien hinzufügen

Das (lokale) Maven-Repository kann auch mit eigenen Dateien oder solchen, die man von dritter Seite erhalten hat, ergänzt werden:

```
mvn install:install-file       \
     -Dfile=Sample.jar         \
     -DgroupId=uniquesample    \
     -DartifactId=sample_jar   \
     -Dversion=2.1.3b2         \
     -Dpackaging=jar           \
     -DgeneratePom=true
```

8.2.10 Software Version Control (SCM)

Mit Hilfe des Plugins `maven-scm-plugin` bietet Maven auch eine einheitliche Schnittstelle zu den Source-Code-Management (SCM) Systemen CVS, SVN, GIT (u.a.) an:

Aktion	Beschreibung
scm:checkin	Änderungen zum SCM-Repository senden
scm:checkout	Arbeitskopie erstellen
scm:add	Datei hinzufügen
scm:update	Dateien aktualisieren
scm:status	Status anzeigen
scm:tag	eine Markierung setzen
...	

Bei einigen Aktionen müssen Parameter angegeben werden:

```
% mvn —Dmessage="<checkin comment here>" scm:checkin
```

Dazu muss in der POM-Datei das Element scm definiert werden:

```
<scm>
  <developerConnection>
    scm:svn:https://somerepository.com/svn_repo/trunk
  </developerConnection>
</scm>
```

Bei GIT lautet die URL wie folgt (Beispiel):

```
scm:git:git://github.com/path_to_repository
```

Weitere Infos: maven.apache.org/scm/git.html

8.2.11 Release-Bildung

Bei der Release-Bildung überprüft das Release-Plugin die folgenden Punkte:

- Sind alle lokalen Änderungen mit dem SCM-Repository abgeglichen?
- Sind alle Integrations-Tests fehlerfrei durchgelaufen?
- Werden bei den verwendeten Bibliotheken und Plugins keine SNAPSHOT-Versionen referenziert?

und sorgt dafür, dass die Versionsbezeichner in den POM-Dateien entsprechend angepasst werden und mit den SCM-Tags korrespondieren:

```
mvn release:prepare —DdryRun=true
```

Das Plugin verlangt anschliessend die folgenden Angaben:

- aktueller Versionsbezeichner (Bsp: 1.0)
- aktueller Releasebezeichner (Bsp: myapp-1.0)
- neuer Bezeichner der Entwicklungsversion (Bsp: 1.1-SNAPSHOT)

und legt die eingebenen Werte in der Datei release.properties ab. Zusätzlich werden verschiedene POM-Dateien erzeugt, die man allesamt mit

```
mvn release:clean
```

wieder löschen muss, wenn Maven die Aktionen ausführen soll.

Nach Eingabe von

```
mvn release:prepare
```

wird Maven den aktuellen Stand des Projektes in das Tags-Verzeichnis kopieren, die neue Versionsbezeichnung in den POM-Dateien einsetzen und mitteilen, dass man nun mit

```
mvn release:perform
```

die eigentlichen Release-Dateien bilden und in das lokale (und Deploy-) Repository transferieren kann. Damit dieser Schritt klappt, muss das distributionManagement-Element definiert sein.

Eine verbesserte Unterstützung bietet das Plugin jgitflow:

```
<plugin>
   <groupId>external.atlassian.jgitflow</groupId>
   <artifactId>jgitflow-maven-plugin</artifactId>
   <version>1.0-m5.1</version>
   <configuration>
     <noDeploy>true</noDeploy>
   <configuration>
</plugin>
```

Plugin goals:

- **jgitflow:release-start** create and push a release branch
- **jgitflow:release-finish** build, tag and merge the release branch into master and develop branches

8.2.12 Exercise

1. Konvertieren Sie das Projekt Histogram in ein Maven-Projekt (artifactId: histogram, groupId=demo) und erstellen Sie die ausführbare Archiv-Datei mit allen benötigten Klassen. Wie lautet der Dateiname der Archiv-Datei, und wie kann man ihn konfigurieren?

8.2.13 Software and further Informations

- Maven: maven.apache.org/
- Maven, The Definitive Guide

  www.sonatype.com/book/reference/public-book.html
- Eclipse und Maven: www.eclipse.org/m2e/

8.3 Gradle

Gradle is a build automation tool for multi-language software development. It controls the development process in the tasks of compilation and packaging to testing, deployment, and publishing. Supported languages include Java (Kotlin, Groovy, Scala), C/C++, and JavaScript.

Gradle builds on the concepts of Apache Ant and Apache Maven, and introduces a Groovy- & Kotlin-based domain-specific language contrasted with the XML-based project configuration used by Maven. Gradle uses a directed acyclic graph to determine the order in which tasks can be run, through providing dependency management.

Gradle was designed for multi-project builds, which can grow to be large. It operates based on a series of build tasks that can run serially or in parallel. Incremental builds are supported by determining the parts of the build tree that are already up to date; any task dependent only on those parts does not need to be re-executed. It also supports caching of build components, potentially across a shared network using the Gradle Build Cache. It produces web-based build visualization called Gradle Build Scans. The software is extensible for new features and programming languages with a plugin subsystem.

Gradle is distributed as open-source software under the Apache License 2.0, and was first released in 2007.

8.4 CMake