

# Medical Software Development

## Master Medical Informatics

Ronald Tanner, David Herzog

FHNW Life Science Technologies, Muttenz

February 2022

- ▶ 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.

- ▶ 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

The final grade will be calculated based on two inputs:

- ▶ 30% Course project
- ▶ 70% Written exam

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

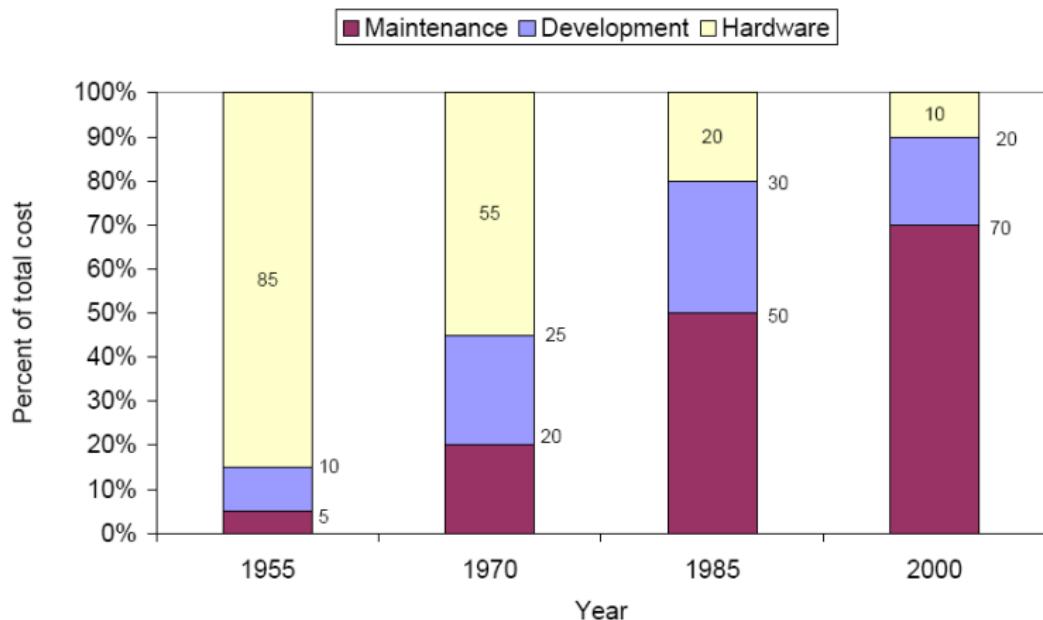
There will be several other frameworks which will be used during the course. These will be installed on demand. The ones listed above should be installed prior the course start.

- ▶ 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](http://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.

Boehm '80, Lientz '87, US DoD '97



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.*

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.*

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*

[www.computer.org/education/bodies-of-knowledge/software-engineering](http://www.computer.org/education/bodies-of-knowledge/software-engineering)

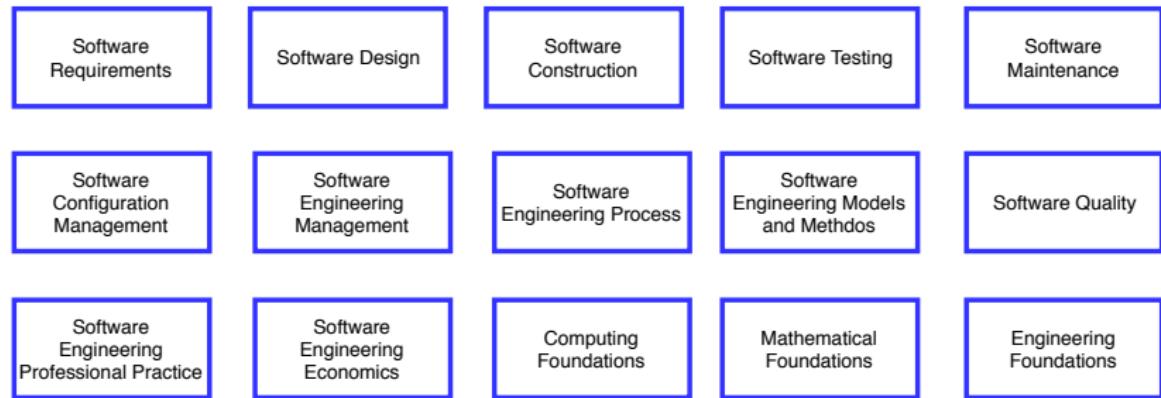
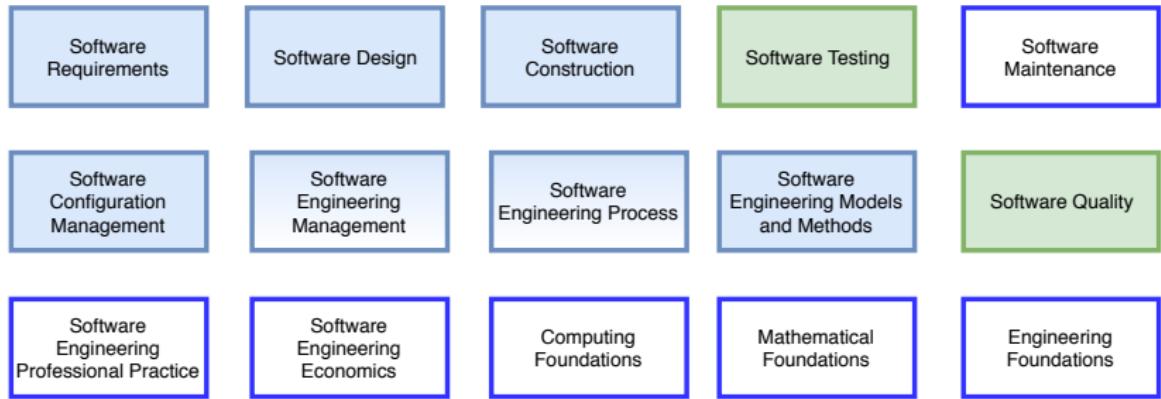


Figure: 15 Knowledge Areas (KA)



- ▶ **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 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

- ▶ **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

- ▶ Computing Foundations: operation systems, networking, algorithms, parallel and 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

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

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](https://en.wikipedia.org/wiki/Pragmatism))

[blog.codinghorror.com/a-pragmatic-quick-reference](http://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.

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

## 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

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>)

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)

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.

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

- ▶ **Data Loader**

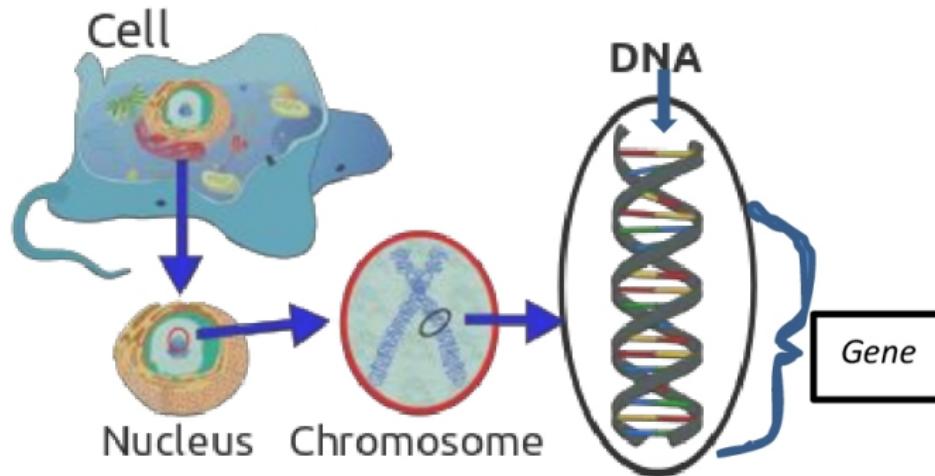
This component loads a data file from NCBI into a database

- ▶ **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

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)



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 and 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.

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?

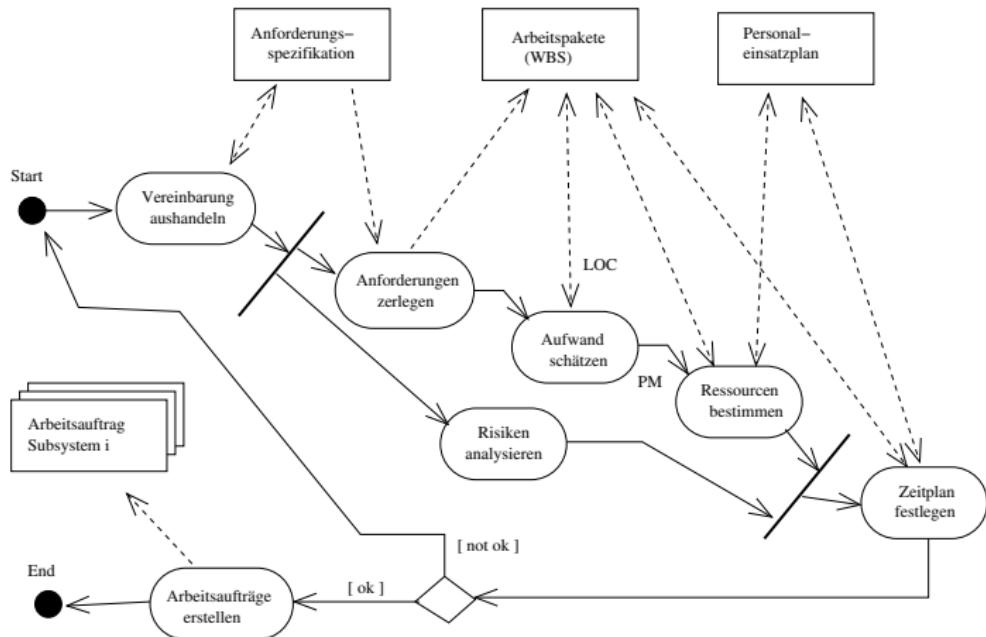
This chapter will cover methods how to control a project. Stakeholders measure projects by how well they are executed within the project constraints or baselines. A project baseline is an approved plan for a portion of a project. It is used to compare actual performance to planned performance and to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

The following 4 project baselines should be considered:

- ▶ Quality
- ▶ Schedule
- ▶ Scope
- ▶ Budget

## Results:

- ▶ **Cost plan:** Creating budget / offer.
- ▶ **Time plan:** Define activities, deadlines and milestones.
- ▶ **Employee plan:** Nomination of people and their working time.
- ▶ **Organization plan:** Agreement of team structure, nomination of responsible persons.
- ▶ **Quality plan:** Compilation of documents, tools and methods to ensure quality.
- ▶ **Project monitor plan:** Agreement of actions to control the project and if needed how to change the plans.
- ▶ **Configuration management plan:** Agreement of actions to control changes in documents, code or data (or any other resources).
- ▶ **Education plan:** Definition of education for internal (project team) and external people (users, operating stuff).
- ▶ **Risk management plan:** Agreement of actions to avoid or mitigate risks.



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

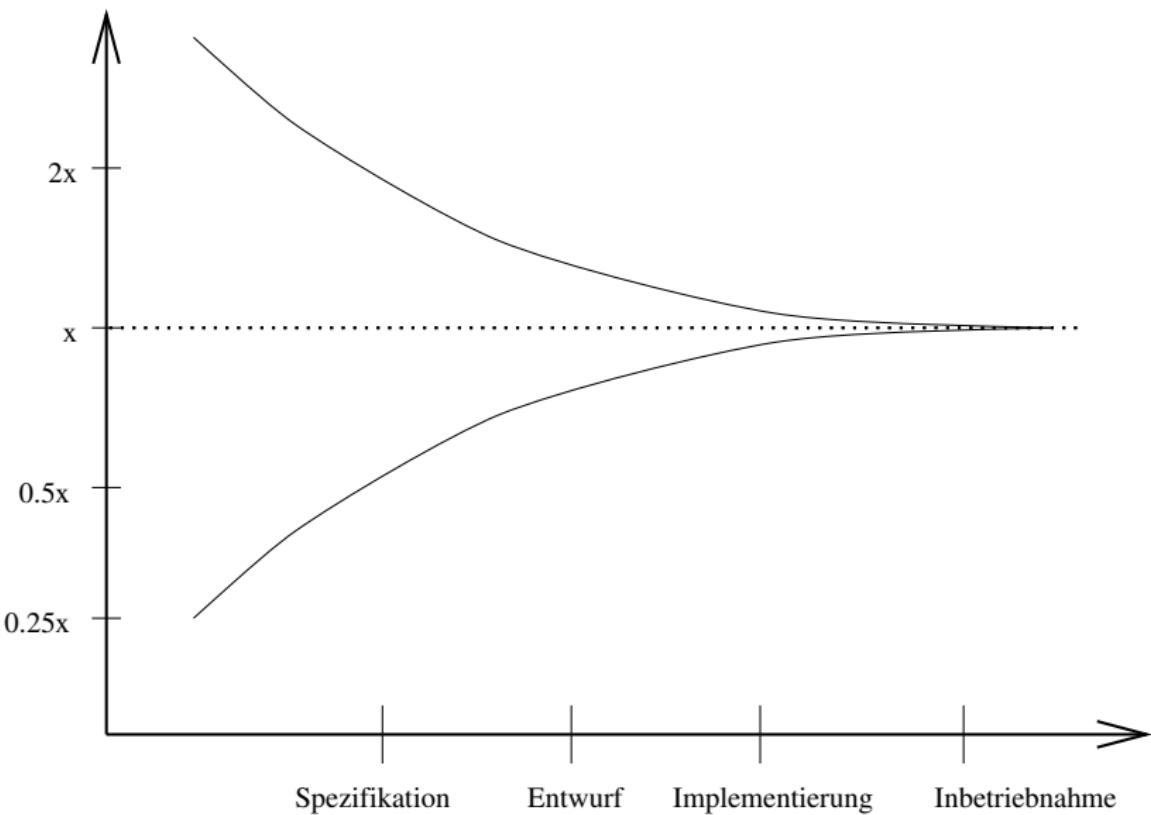
- ▶ **MVP**: Minimal viable product. How should the problem be solved to bring additional value. What is too much, what is not enough?
- ▶ **Priority**: Which tasks should be executed first?
- ▶ **Release**: Which pieces should be part in the next release?
- ▶ **Delivery**: Which timeframe is available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

The total costs of software projects contains

- ▶ **Software-Cost:** software, licenses
- ▶ **Hardware-Cost:** infrastructure (own hardware, cloud-based systems)
- ▶ **Personal-Cost:** Salary, Expenses



Influencing factors to estimate the costs:

- ▶ **Size:** Lines of code, classes, methods
- ▶ **Complexity:** Dependency between the modules, is it possible to build modules?, HW/SW environment
- ▶ **Experience:** Number of similar executed projects
- ▶ **Reliability:** error rate, system stability

- ▶ 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 are used to compute a functional size measurement (FSM) of software.: ([www.ifpug.org](http://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)

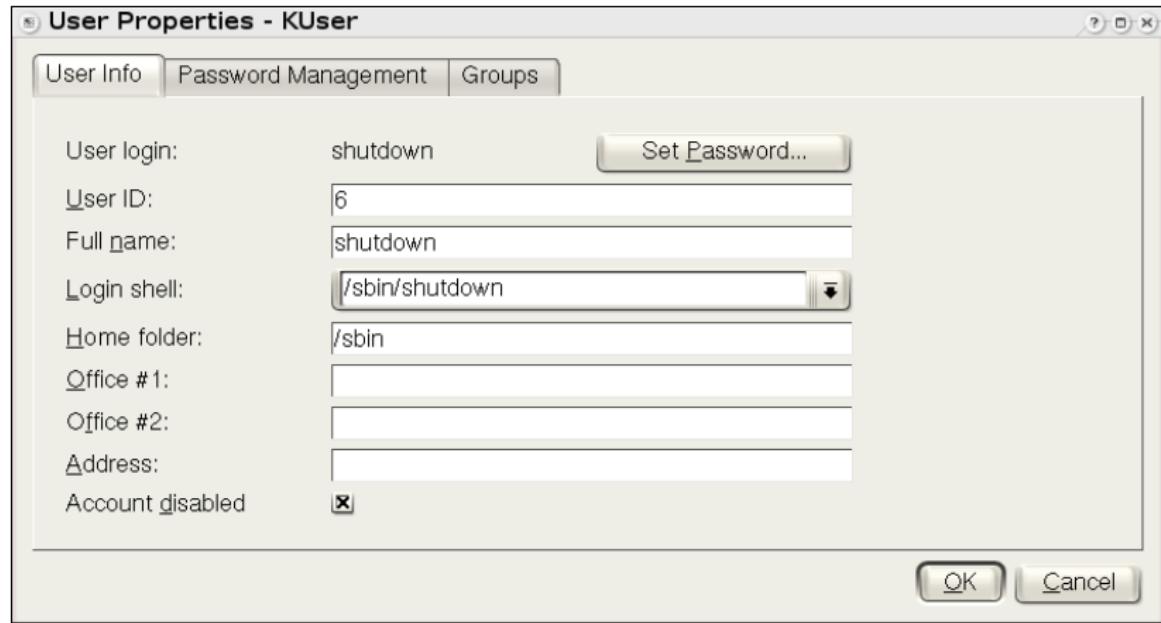
Calculate the number of function points based on the elements of the user interface.

$$\text{functionpoints} = 2 \cdot \text{widgetpoints} \quad (1)$$

- ▶ **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

# Widget Points Example

53/418



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

1. Calculation of the effort in person month:

$$E_i = a \cdot KDL^b \quad (2)$$

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:

$$E = EAF \cdot E_i \quad (3)$$

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

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

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

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.

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.

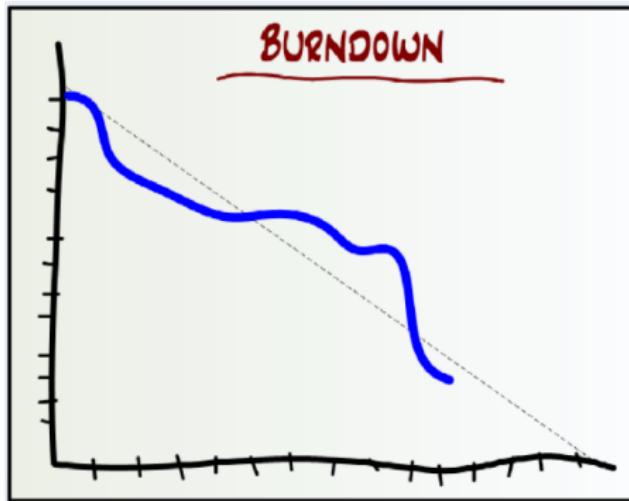


Figure: Burndown chart (Quelle: Henrik Kniberg)

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.

Axiom: persons and months are NOT exchangeable!

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

$$D = 2.5 \cdot E^{0.38} \quad (4)$$

2. Calculate the average demand of employees:

$$P = \frac{E}{D} \quad (5)$$

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

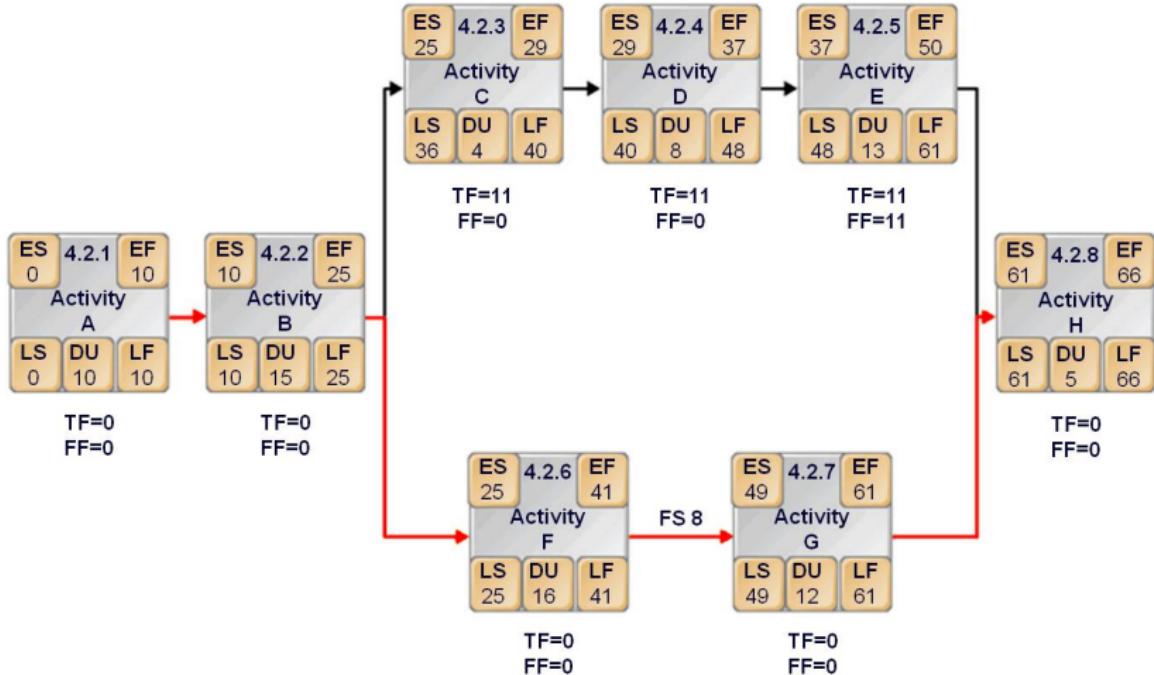
Software size	small (2 KDL)	medium (32 KDL)	large (128 KDL)
Design	19	19	19
Implementation	63	55	55
Integration + Tests	18	26	26

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	...

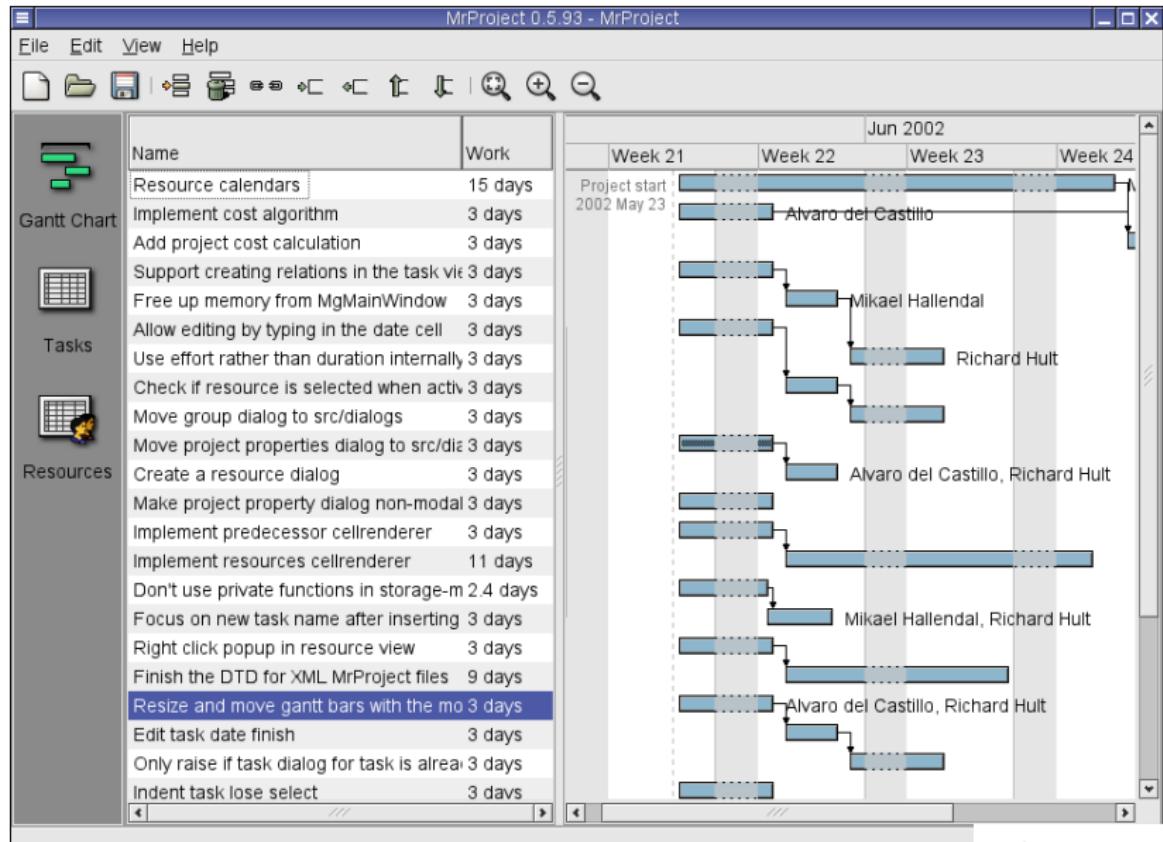
# Critical Path Method / Critical Path Analysis

64/418



# Gantt Diagram

65/418



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 of receiving frequent customer feedback, stakeholders or project 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, 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 component delivery by 3rd party vendor
Financial issue	Reorganization, incorrect budget estimation, project scope expansion, cost overruns
Communication issues	Inadequate support, misunderstanding, conflicts



$$Risk = Likelihood \cdot Severity \quad (6)$$

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 occurrence: 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 occurrence in the first place.
- ▶ Severity: Reduce the impact of the risk on the critical success factors of the project.

## 5. Transference

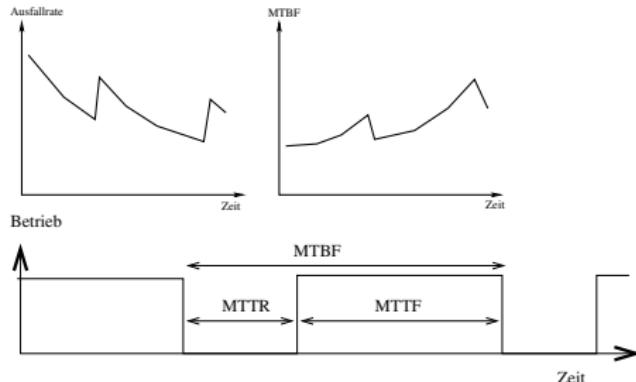
Finally, you can transfer the risk onto another party.

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.

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



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



failure	Deviation of the operating behavior from its requirements
fault, bug	Cause of one or more malfunctions
error	wrong, incorrect or incomplete implementation
reliability	Probability of trouble-free operation during a certain period of time
availability	Ratio of the trouble-free operating time to the total operating time

MTBF	mean time between failures	Nines of Reliability:	downtime per year		
			[h]	[min]	[sec]
MTTR	mean time to repair	2 9's (99%)	87.6	5256.0	315360.0
MTTF	mean time	3 9's (99.9%)	8.76	525.1	n w
		4 9's (99.99%)	0.876	52.5	Fachhochschule Nordwestschweiz

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.

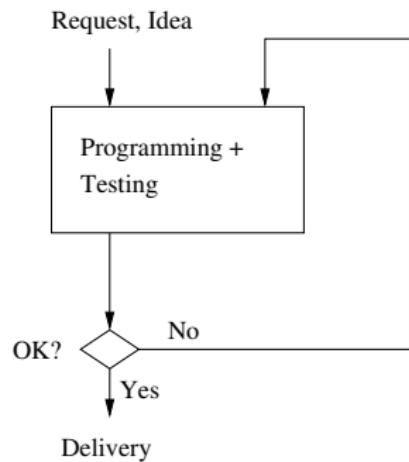
The software development life-cycle (SDLC) emphasizes the key activities and their temporal and logical interdependencies and relationships.

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

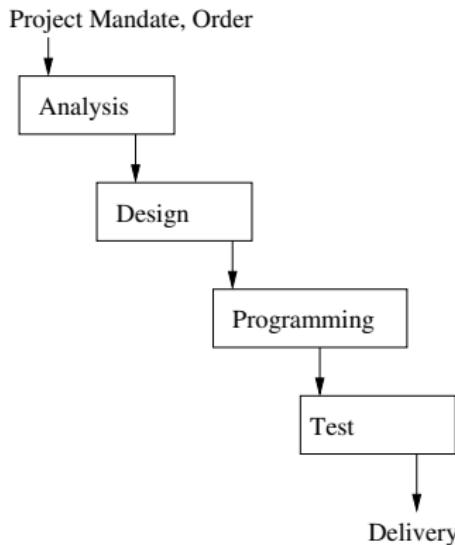
## Adhoc Model:

spontaneous, individual process with little or no control:

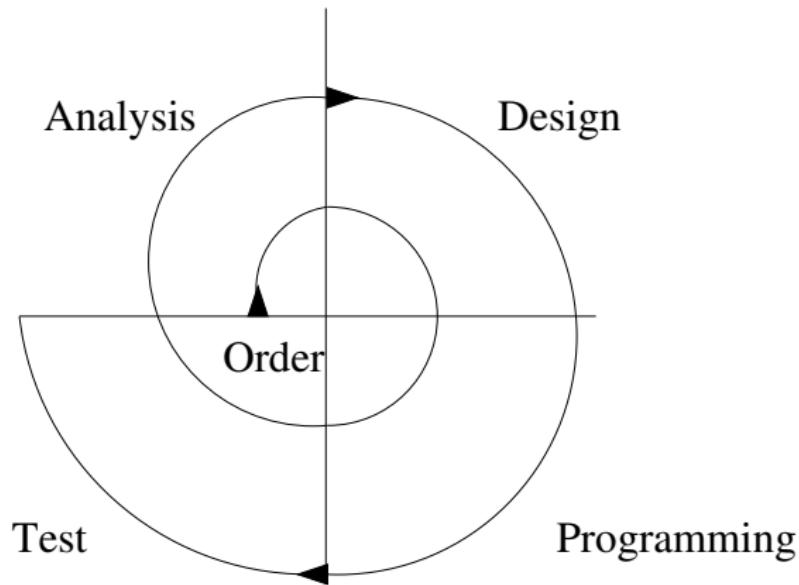


## Waterfall Model:

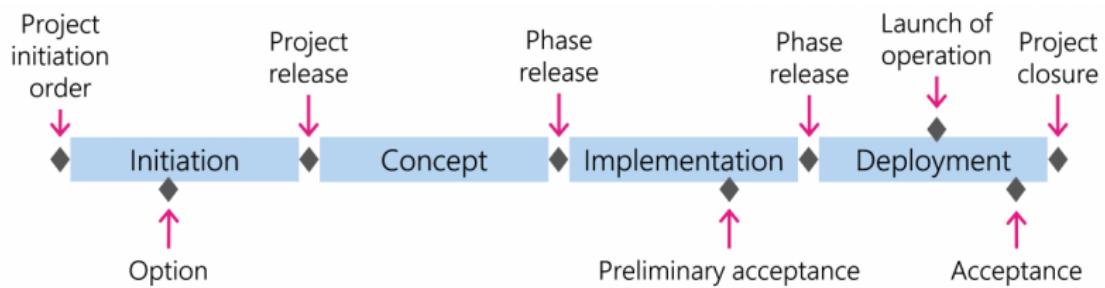
strictly sequential, immutable process



cyclic, risk oriented



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](http://www.hermes.admin.ch)



**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

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

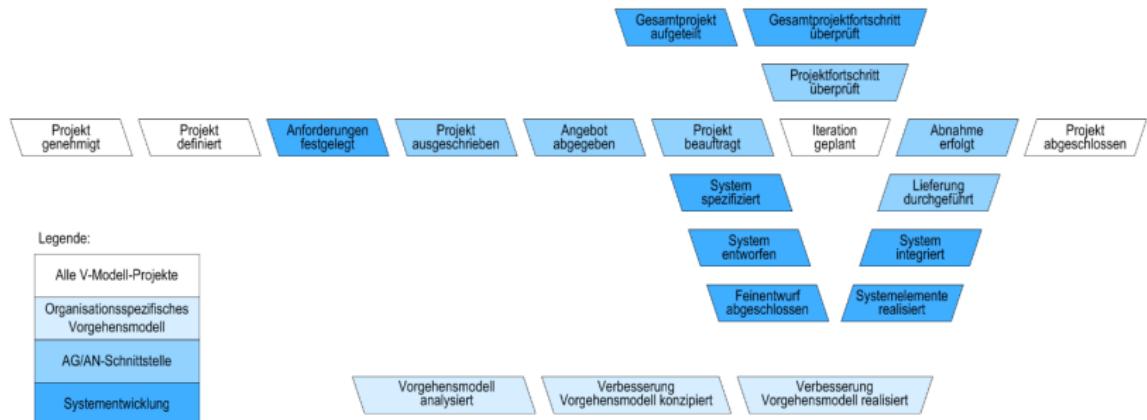
- ▶ Project steering
- ▶ Project management
- ▶ Agile development
- ▶ Project foundations
- ▶ Business organization
- ▶ Product
- ▶ IT system
- ▶ Procurement
- ▶ Deployment
- ▶ organization
- ▶ Testing
- ▶ IT migration
- ▶ IT operation
- ▶ Information security and data protection

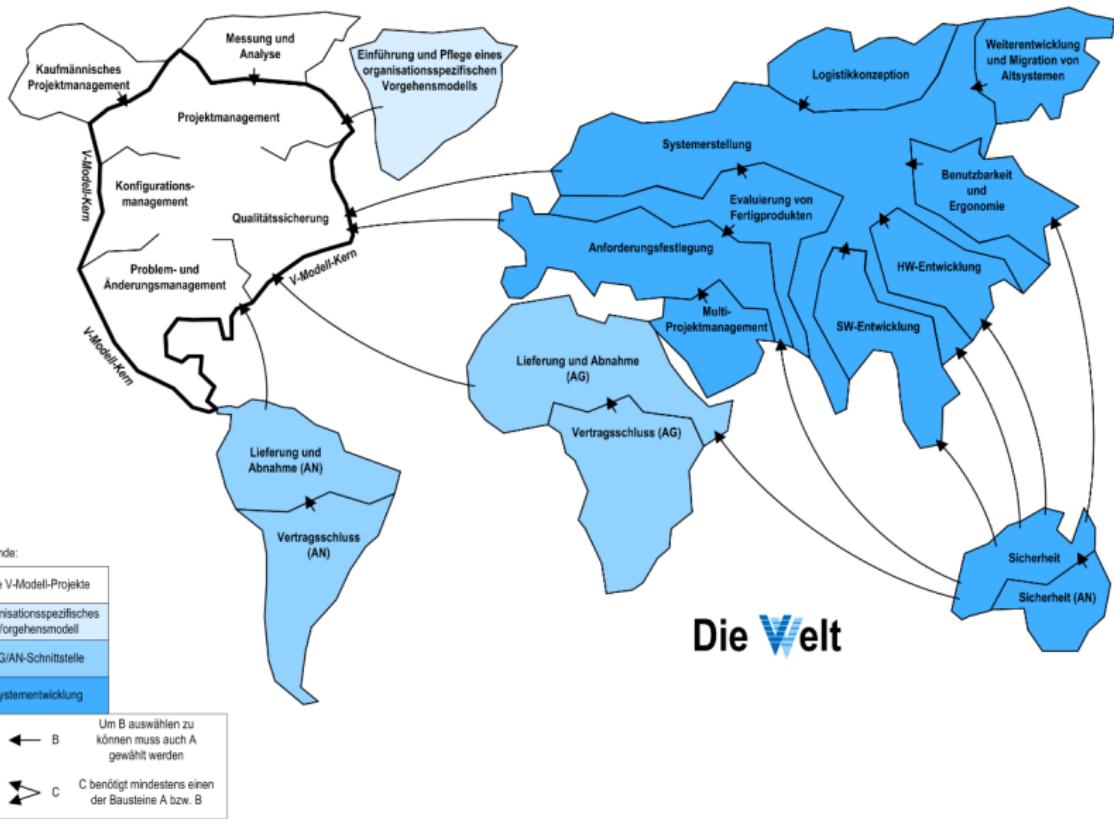
Specific Modules: Marketing, Communication, Personnel development, Training, Strategy development, Business administration, Deployment

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](http://www.v-modell-xt.de))

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

- ▶ 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

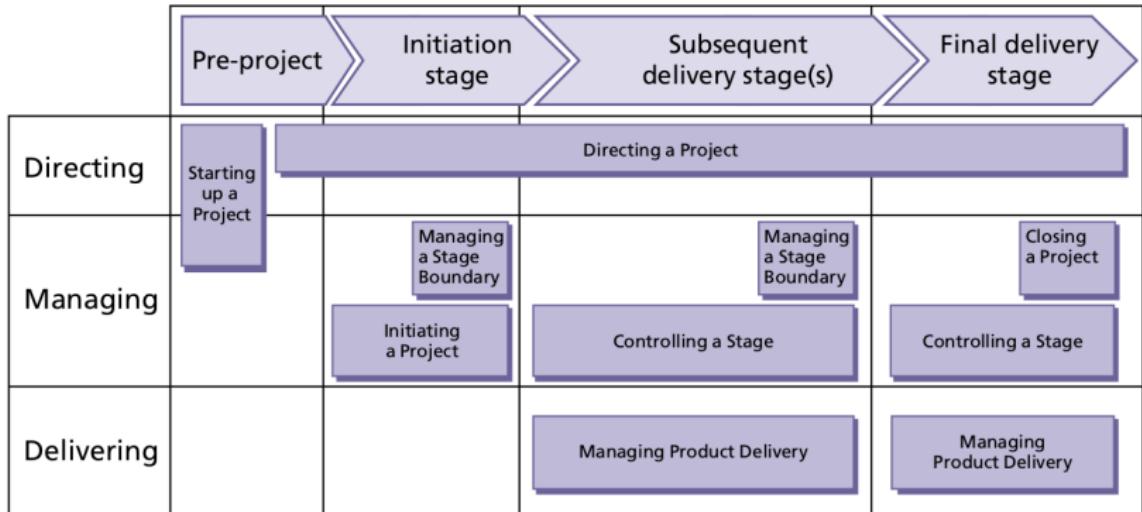




PRINCE2 (PRojects IN Controlled Environments) is a structured project management method and practitioner certification programme developed by the UK government.

- ▶ Seven Themes: Business Case, Organisation, Quality, Plans, Risk, Change, Progress
- ▶ Seven Principles: Continuous 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

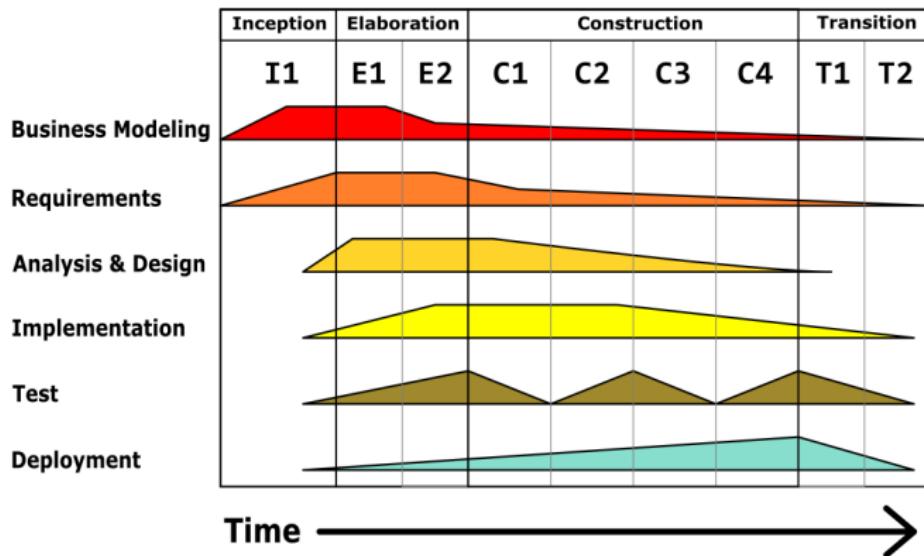
[www.axelos.com/best-practice-solutions/prince2/what-is-prince2](http://www.axelos.com/best-practice-solutions/prince2/what-is-prince2)



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

## Iterative Development

Business value is delivered incrementally in time-boxed crossdiscipline iterations.



**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.

- 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

**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,

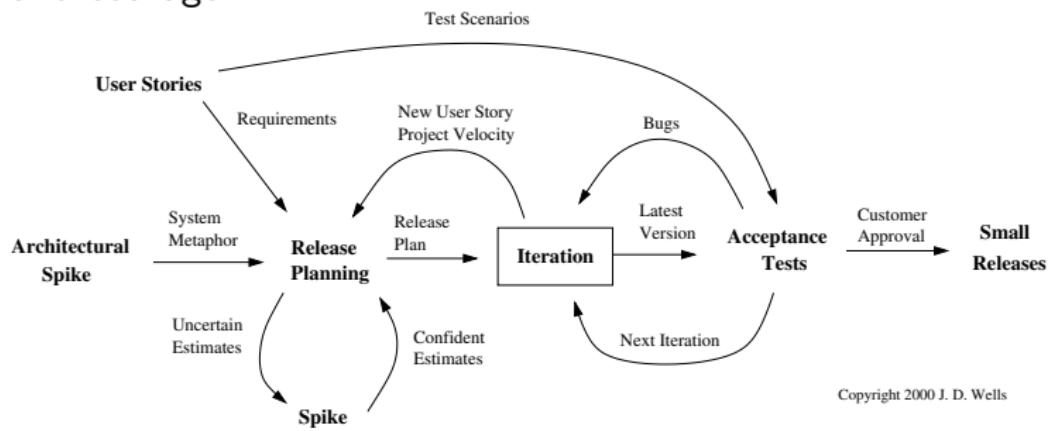
**Test:** planning, specifying, designing, executing system tests

**Deployment:** delivery, installation and putting into service

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
Post condition	state of application after everything is done

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](http://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 around.

## 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.

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).

- ▶ 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.

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	<ol style="list-style-type: none"><li>1. User selects 'Catalogs' menu item.</li><li>2. System displays criteria for search and list of catalogs present in the system.</li><li>3. User clicks on "New Prod Catalog" button.</li><li>4. System displays a form to create the new catalog.</li><li>5. User enters the information and click on "Update" button.</li><li>6. System successfully creates a new catalog.</li><li>7. User again selects 'Catalogs' menu item.</li><li>8. User Clicks on any catalog Id.</li><li>9. System displays edit product catalog page.</li><li>10. User enters/updates the information and click on "Update" button.</li><li>11. System successfully updates the catalog.</li></ol>
Post condition	User is able to Create/Edit catalog.

---

## 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.

---

THE PRIVILEGED ADMINISTRATOR - Your Company Name Here Language : English | Visual Themes | Logout |

Applications > Catalog Manager > **Find Catalog**

**Search Products**

Keywords:   
Category ID:    
No Contains  Any  All   
Category ID:    
[Advanced Search](#)    
-Product Jump-

**New Prod Catalog**

**Search Options**

ProdCatalogId Contains   Ignore Case  
Catalog Name Contains   Ignore Case  
**Find**

**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

**Browse Catalogs/Categories**

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

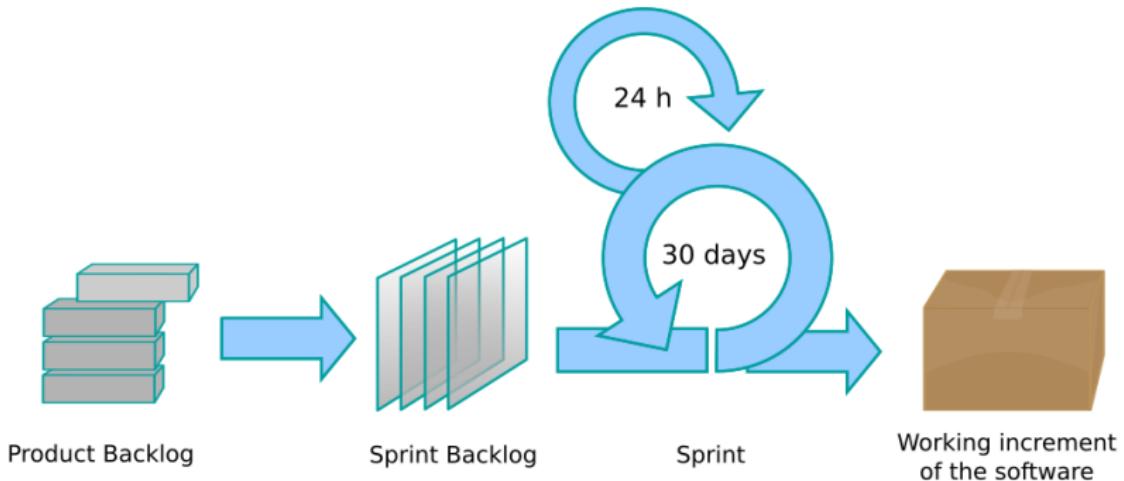
I	Independent	The user story should be self-contained, in a way that there is no inherent dependency on another user story.
N	Negotiable	User stories, up until they are part of an iteration, can always be changed and rewritten.
V	Valuable	A user story must deliver value to the end user.
E	Estimable	You 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.

Scrum<sup>1</sup> 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 development 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.



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?

# A Scrum Task Board

106/418

Story	To Do	In Process	To Verify	Done
As a user, I... 8 points	Code the... 9  Code the... 2  Test the... 8	Test the... 8  Code the... 8  Test the... 4	Code the... DC 4  Test the... SC 8	Test the... SC 6  Code the... D Test the... SC 8 Test the... SC Test the... SC Test the... SC 6
As a user, I... 5 points	Code the... 8  Code the... 4	Test the... 8  Code the... 6	Code the... DC 8	Test the... SC Test the... SC Test the... SC 6

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

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](http://manifesto.softwarecraftsmanship.org)

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

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.

## *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.*

**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.

**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

- ▶ **Who** is involved?  
(User profiles, Stakeholders, Responsibilities, Organization)
- ▶ **What** is the current state?  
(Business processes, problems, weaknesses, opportunities)
- ▶ **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.)

- ▶ 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

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

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](https://en.wikipedia.org/wiki/Non-functional_requirement)

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.

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*

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

- ▶ complete
- ▶ unambiguous
- ▶ correct
- ▶ traceable
- ▶ consistent
- ▶ prioritised
- ▶ verifiable
- ▶ feasible

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.

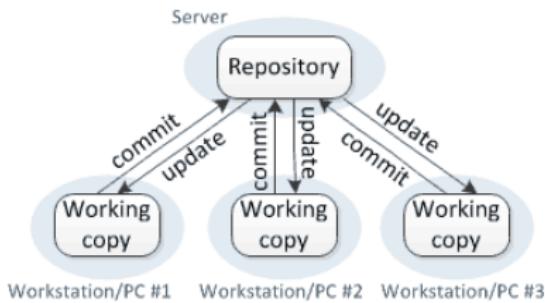
A version control system serves the following purposes, among others:

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.

- ▶ 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.

## Centralized version control



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

- ▶ You commit
- ▶ They update

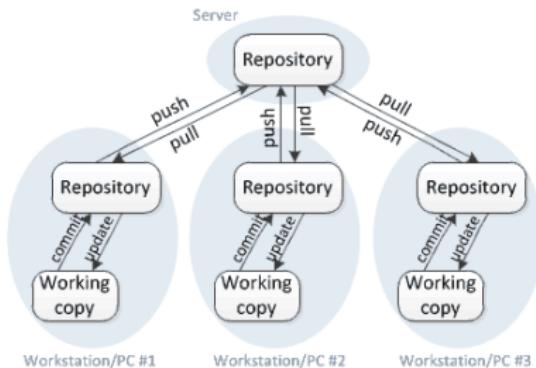
Typical systems are:

- ▶ 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.

Distributed version control



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

## 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.

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

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:

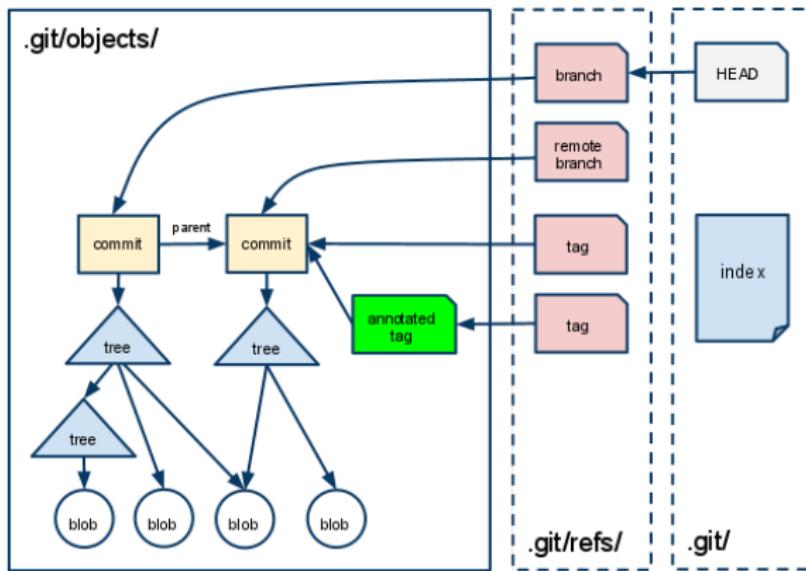
- ▶ **Blob**: A blob is used to store file data. It is generally a file.
- ▶ **Tree**: A tree is basically like a directory. It references a bunch of other trees and blobs
- ▶ **Commit**: A commit object holds metadata for each change introduced in the repository, including the author, committer, commit-data, 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:

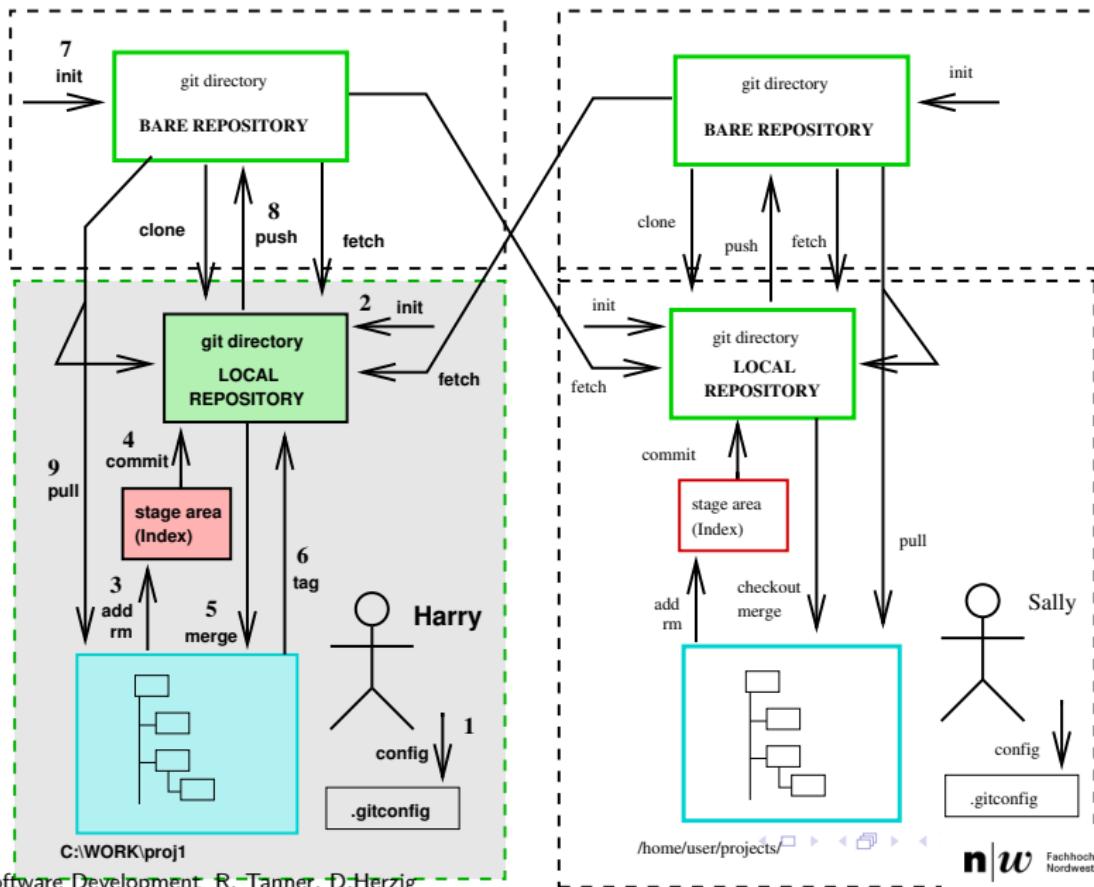
- ▶ executing experiments
- ▶ try out new ideas
- ▶ development of new features

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.

# Git Commands

138/418



## 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:

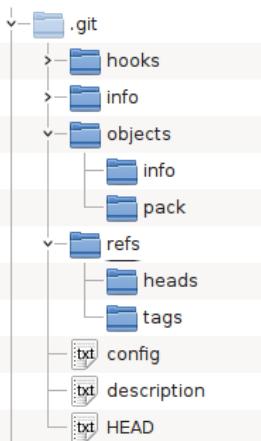
- ▶ User level
- ▶ Repository level
- ▶ System level

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/proje...
```



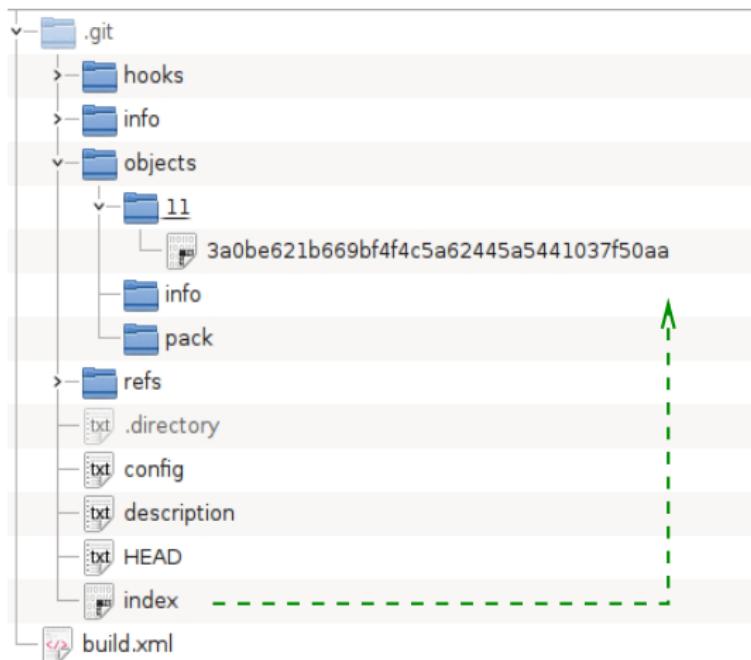
**Note:** To create a local repository from an existing one, the command `init` needs to be replaced with `clone`.

```
$ 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.

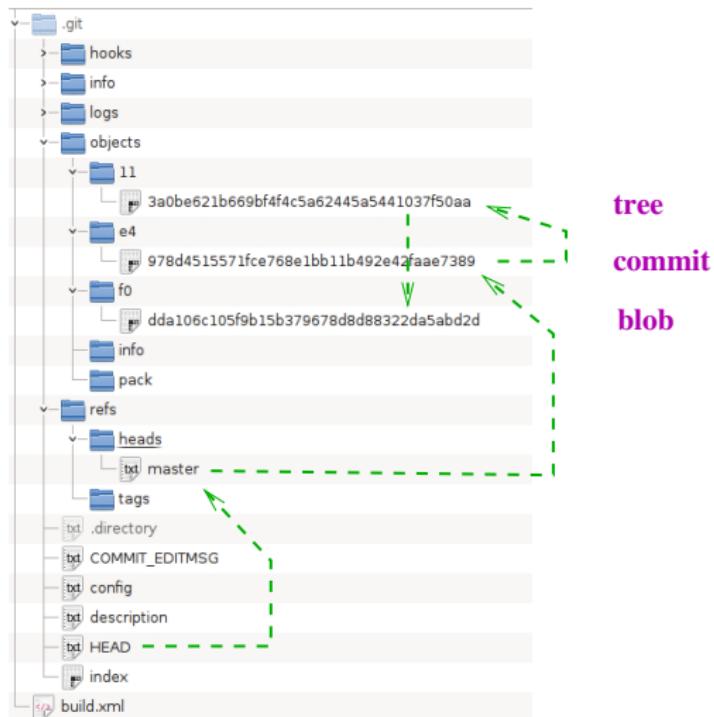


4. commit: Record the changes made to the files to a local repository. For easy reference, each commit has a unique ID:

```
$ 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"
```



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 addition:

```
$ 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.

- ▶ 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 --since=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
```

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
```

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/proj'
To prevent you from losing history, non-fast-forward upda
Merge the remote changes (e.g. 'git pull') before pushin
See the
'Note about fast-forwards' section of 'git push --help'
```

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 th
```

The lines with conflicts will be marked in the file:

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

These conflicts needs to be fixed manually.

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.

**C**omplete **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

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 file 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 <b>compile</b> 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.

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

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>
```

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.

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>
```

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

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.

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.

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:testReso
4 test-compile	maven-compiler-plugin	compiler:testComp
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

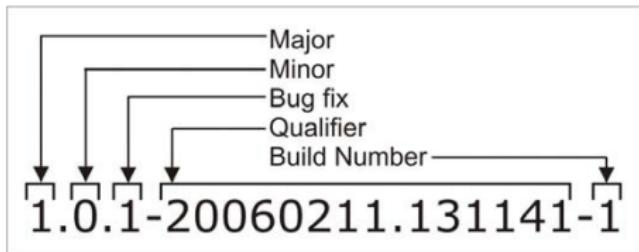
A simple example will explain how Maven could be used:

## 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:** 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:

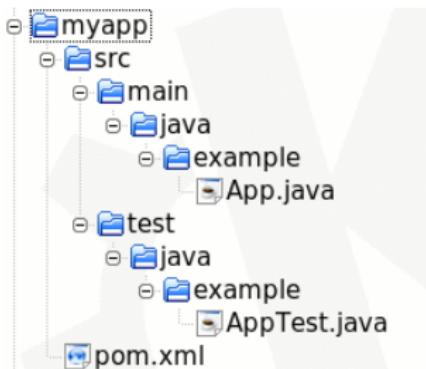


Figure: Directory structure of a Maven project

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](http://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>

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

- 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.

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.

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
```

Variables can be included in your resources. These variables, denoted by the \${...} delimiters, can come from the system properties, your project properties, from your filter resources and from the command line.

```
ResourceBundle rb =  
    ResourceBundle.getBundle("ApplicationResources");  
  
String appname = rb.getString("application.name");  
String version = rb.getString("application.version");
```

A file called ApplicationResources.properties should now be stored in src/main/resources|. This file will contain

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

Maven will now use the values defined in the POM file and will replace project name and project version.

For that, the resource element needs to be configured:

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

The execution of automated unit tests is an important part in the build process. Execution of unit tests is also possible with Maven.

```
mvn test
```

The result will be stored as report in target/surefire-reports.  
The behaviour of the plugin could be configured:

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

```
mvn site
```

This command will create a project documentation. This documentation is stored in target/site.

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

The layout of the created documentation could be configured with:  
src/site/site.xml

# Omnis Litteris

Last Published: 2009-01-18

Omnis Litteris

## Project Documentation

### ▼ Project Information

#### About

Continuous  
Integration  
Dependencies  
Issue Tracking  
Mailing Lists  
Project License  
Project Summary  
Project Team  
Source Repository

### ► Project Reports



## 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: A Maven generated project page

There are several plugins available for a full documentation:

- ▶ 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>
```

The local Maven repository could be extended with libraries which are not available in public.

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

The plugin `maven-scm-plugin` is an interface to version control systems:

Action	Description
<code>scm:checkin</code>	commit changes
<code>scm:checkout</code>	checkout
<code>scm:add</code>	add file
<code>scm:update</code>	update file
<code>scm:status</code>	show status
<code>scm:tag</code>	create tag
...	

Parameters needs to be provided for some actions:

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

In addition, the scm element needs to be defined with in the POM file.

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

Example for a git repository

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

Additional information: [maven.apache.org/scm/git.html](http://maven.apache.org/scm/git.html)

While creating a release, the release-plugin is checking the following points:

- ▶ Are all local changes in sync with the SCM repository?
- ▶ Are all integration tests successfully executed?
- ▶ Are no SNAPSHOT versions in use for libraries and plugins?

Also checks if the version in the POM is correct and in sync with SCM tags.

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

The following information needs to be provided:

- ▶ current version (e.g. 1.0)
- ▶ current release (e.g. myapp-1.0)
- ▶ new version (e.g. 1.1-SNAPSHOT)

All values are stored in `release.properties`. In addition, several POM files will be created. These need to be deleted with

```
mvn release:clean
```

After

```
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

1. Create a Maven project for the Gene REST Service application. The REST interface should provide the following 3 GET operations:
  - ▶ Search by ID  
Exact match with the gene identifier
  - ▶ Search by Symbol  
Exact match with the gene symbol
  - ▶ Search by Description  
If the provided search term could be found in the description
2. Add 2 profiles with the database URL to the project
3. Connect the project with the source repository (pom.xml)

- ▶ Maven: [maven.apache.org/](http://maven.apache.org/)
- ▶ Maven, The Definitive Guide  
[www.sonatype.com/book/reference/public-book.html](http://www.sonatype.com/book/reference/public-book.html)
- ▶ Eclipse und Maven: [www.eclipse.org/m2e/](http://www.eclipse.org/m2e/)

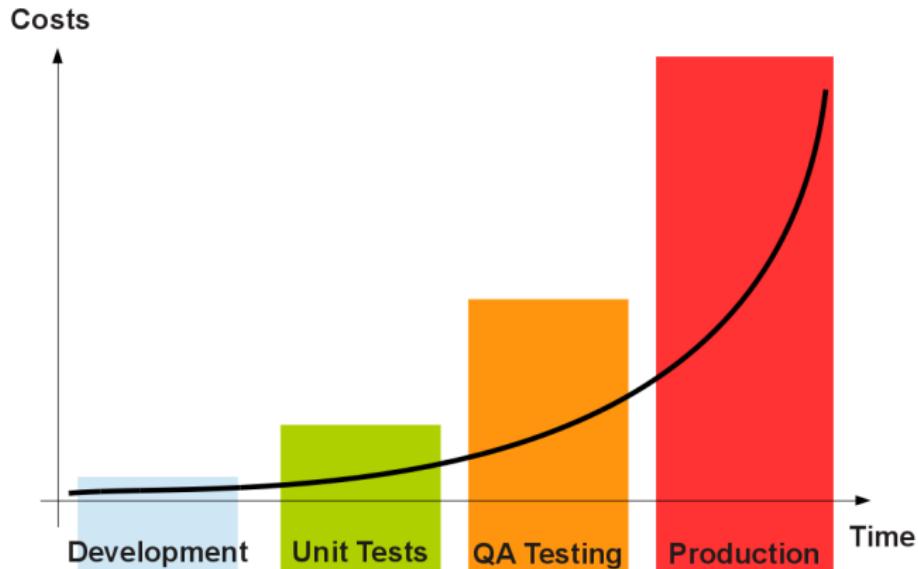


Figure: Bug Fixing Costs

*We are our own best testing team and should never allow anyone else to experience bugs or see the software crash. Do not waste others time. Test thoroughly before checking in your code.*

*Great tools help make great software. Use available tools whenever it makes sense.*

- ▶ Unit-Test-Framework
  - supports the creation and execution of unit tests
- ▶ Automatic execution
  - After definition and implementation of a set of test cases, these test cases could be repeated in an automatic fashion, without having any user interaction (repeat/playback).
- ▶ Source code analysis
  - Monitoring complexity and compliance with coding conventions.
- ▶ Coverage reports
  - Create a report with the information about how much of the code is covered with a unit test.
- ▶ Memory analysis
  - Analyzing the memory usage of the software.

- ▶ Load test

Analyze the behavior of the system with a different number of simultaneous executed operations (load/performance test).

- ▶ Web tests

Checking functionality, usability, security, compatibility and performance of web application.

- ▶ Management and documentation

Test planning and documentation

Source code analysis is one of the most thorough methods available for auditing software. A scanner is used to find potential trouble spots in source code, and then these spots are manually audited for security concerns.

Typical categories are:

- ▶ Compliance with coding guidelines (code conventions)
- ▶ Object comparison
- ▶ Equals / hashCode issues
- ▶ Performance issues
- ▶ Unused local variables, methods, parameters
- ▶ Duplicated code

FindBugs is an open source tool used to perform static analysis on Java code.

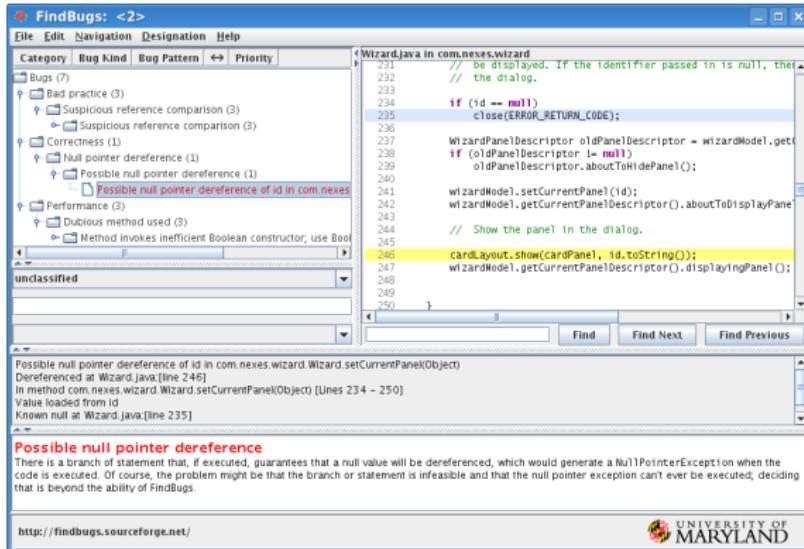


Figure: FindBugs User Interface

PMD is a source code analyzer. It finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth.

The screenshot shows a web browser window displaying the PMD Results for a project named 'aproject'. The URL is <http://localhost/sites/aproject/pmd.html>. The page has a header with the project name and a 'Last Published' date of 2008-02-23. On the left, there's a sidebar with 'Project Documentation' links: Project Information, Project Reports, CPD Report, JavaDocs, PMD Report, Source Xref, Test JavaDocs, and Test Source Xref. Below that is a 'Built by Maven' logo. The main content area is titled 'PMD Results' and contains the message: 'The following document contains the results of PMD 4.1.' A 'Files' section lists 'com/maventest/App.java'. Under the 'Violation' section, two items are listed:

Violation	Line
Avoid unused local variables such as 'x'.	12
Avoid unused private methods such as 'doNothing()'	15

At the bottom, there's a footer with the text 'How do I generate PMD and CPD reports for a site?' and a copyright notice: © 2008.

Figure: PMD Report

A PMD report could be created within the Maven process:

```
<project>
  ...
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-pmd-plugin</artifactId>
      <version>3.16.0</version>
    </plugin>
  </plugins>
</build>
  ...
</project>
```

The report is generated by executing the Maven goal `mvn pmd:check`. The report `pmd.html` is available in `target/site`. The Maven process will fail if there is one or more coding violations (according to the PMD rules).

The plugin could also be included in the <reporting> section:

```
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-pmd-plugin</artifactId>
        <version>3.16.0</version>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

The check will then be executed with the `mvn site` goal.

The rules are available under

[https://pmd.github.io/latest/pmd\\_rules\\_java.html](https://pmd.github.io/latest/pmd_rules_java.html)

SonarQube is a Code Quality Assurance tool that collects and analyzes source code, and provides reports for the code quality of your project. It combines static and dynamic analysis tools and enables quality to be measured continually over time.

<https://www.sonarqube.org/>

```
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.sonarsource.scanner.maven</groupId>
        <artifactId>sonar-maven-plugin</artifactId>
        <version>3.9.1.2184</version>
      </plugin>
    </plugins>
  </pluginManagement>
</build>
```

Start Sonar and create a project and an access token.

```
mvn sonar:sonar -Dsonar.host.url=http://localhost:9000
-Dsonar.login=the-generated-token
```

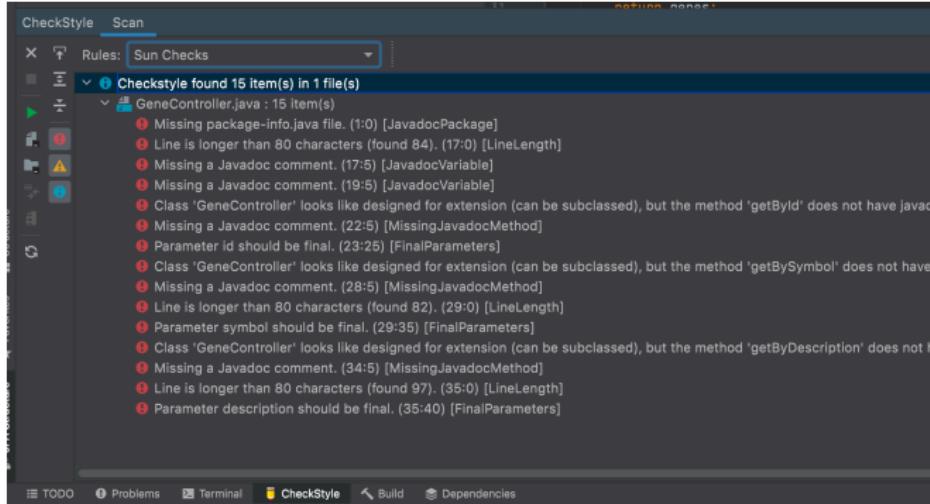
Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans of this boring (but important) task. This makes it ideal for projects that want to enforce a coding standard.

Checkstyle could be integrated in all major development environments (e.g. IntelliJ, Eclipse, Netbeans). It is also possible to use Checkstyle as an independent tool:

```
% java -jar checkstyle-10.2.jar \
    -c sun_checks.xml
    src
```

In this example the code located in the `src` directory will be checked against the rules defined in the `sun_checks.xml` file.

## With IntelliJ IDE:



With Ant:

```
<target name="checkstyle">
    <checkstyle config="checkstyle-10.2/sun_checks.
        xml">
        <fileset dir="src" includes="**/*.java" />
    </checkstyle>
</target>
```

With Maven:

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</
          artifactId>
      <configuration>config/sun_checks.xml</
          configuration>
    </plugin>
  </plugins>
</reporting>
```

The checkstyle report will then be generated with mvn site.

With Eclipse:

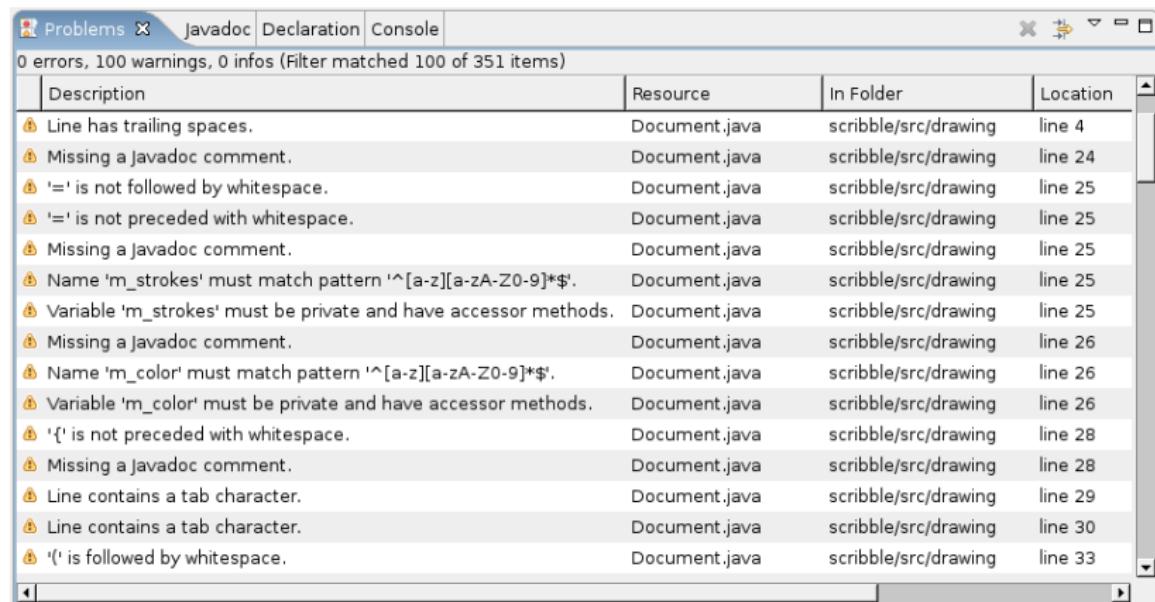


Figure: Checkstyle-Eclipse-Plugin

An assertion is a statement in Java which ensures the correctness of any assumptions which have been done in the program. When an assertion is executed, it is assumed to be true. If the assertion is false, the Java Virtual Machine will throw an Assertion error. It finds its application primarily in the testing purposes. Assertion statements are used along with boolean expressions.

```
double d = 4 * a * c - b*b;  
assert d >= 0 : "Negative Value";  
double x1 = sqrt( d );
```

By default, assertions are disabled in Java. In order to enable them we use the vm parameter ea:

```
% java -ea AssertionTest
```

It is important to know the memory consumption behavior of the software system. Especially to find out if there are any memory leaks.

A memory leak is a type of resource leak that occurs when a computer program incorrectly manages memory allocations in a way that memory which is no longer needed is not released. After a specific amount of time, the system will crash with a memory exception.

A hint for a memory leak is the increase of memory usage over time.

There are several tools available to check for memory usage.

Default tools are:

- ▶ Windows: Task Manager
- ▶ Linux/OSX: top command

# Memory Analysis II

230/418

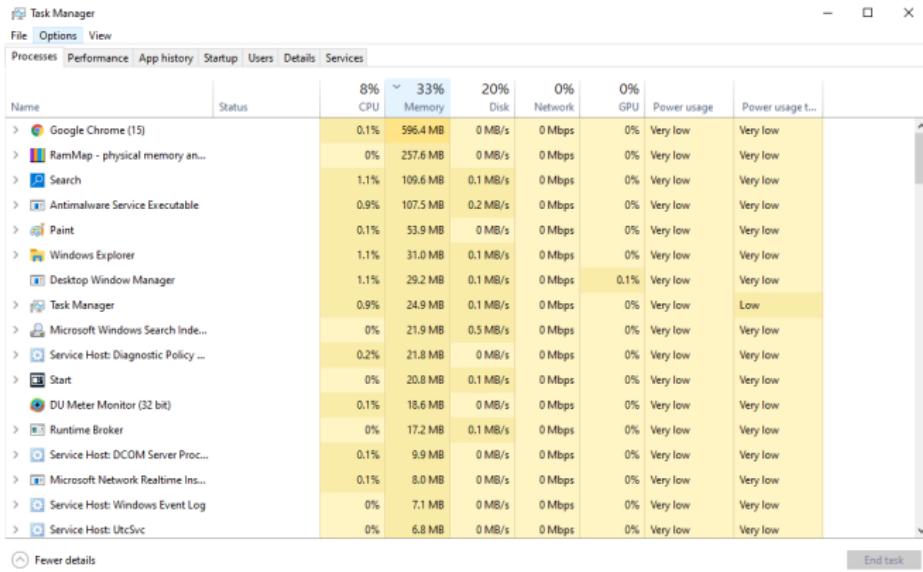


Figure: Windows Task Manager

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1	root	20	0	37168	1192	680	S	0.0	0.2	2:21.51	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd/646
3	root	20	0	0	0	0	S	0.0	0.0	0:00.01	khelper/646
62	root	20	0	38896	1316	1188	S	0.0	0.3	1:08.85	systemd-journal
219	root	20	0	26012	328	200	S	0.0	0.1	0:11.09	cron
226	root	20	0	65464	924	220	S	0.0	0.2	0:13.11	sshd
229	syslog	20	0	184632	1524	464	S	0.0	0.3	0:28.85	rsyslogd
231	root	20	0	47572	504	40	S	0.0	0.1	0:07.80	rpcbind
274	root	20	0	12788	8	4	S	0.0	0.0	0:00.00	agetty
275	root	20	0	12788	8	4	S	0.0	0.0	0:00.00	agetty
293	root	20	0	308984	12800	2248	S	0.0	2.4	27:15.96	fail2ban-server
4452	root	20	0	92996	3124	3120	S	0.0	0.6	0:00.03	sshd
4461	supriyo	20	0	92996	1000	996	S	0.0	0.2	0:00.00	sshd
4462	supriyo	20	0	19472	1604	1600	S	0.0	0.3	0:00.05	bash
4696	root	20	0	92996	4036	3132	S	0.0	0.8	0:00.02	sshd
4705	supriyo	20	0	92996	1952	1008	S	0.0	0.4	0:00.02	sshd
4706	supriyo	20	0	19472	3364	1600	S	0.0	0.6	0:00.05	bash
4718	supriyo	20	0	36608	1784	1324	R	0.0	0.3	0:00.31	top
5830	root	20	0	41532	728	320	S	0.0	0.1	0:01.25	systemd-udevd
13879	www-data	20	0	290032	2632	2612	S	0.0	0.5	0:01.18	php-fpm7.0
14031	cloud-t+	20	0	19788	9736	3276	S	0.0	1.9	10:11.27	cloud-torrent
14089	root	20	0	286068	560	452	S	0.0	0.1	1:11.67	php-fpm7.0
14091	www-data	20	0	289508	2168	2064	S	0.0	0.4	0:02.32	php-fpm7.0

Figure: top command

Zulu Mission Control is a build of JDK Mission Control, an open source Java runtime profiling and monitoring utility originally developed by JRockit and open source by Oracle in 2018. JDK Mission control was originally developed for Java 11, and Azul Systems has back-ported the utility to also support properly-configured builds of OpenJDK 8.

Zulu Mission Control will detect a running Java application and will be able to hook up with Java application.

# Memory Analysis V

233/418

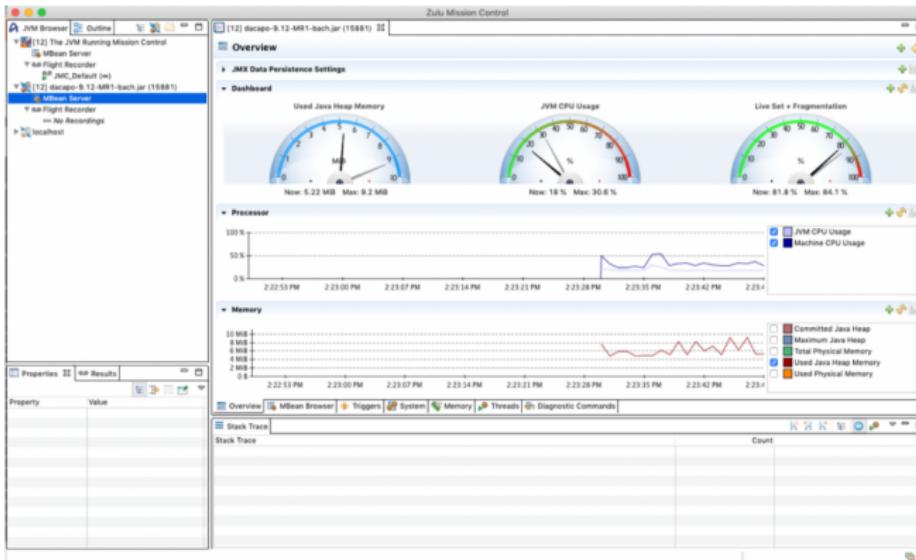


Figure: Zulu Mission Control

<https://www.azul.com/products/zulu-mission-control/>

A large part of software developers lives are monitoring, troubleshooting and debugging.

The possibility to reproduce a bug on the local development machine makes it easier to find the root cause. A debugger could be used to walk through the code, step by step.

If it is not possible to reproduce the bug on the local development environment makes it very hard.

Proper logging makes this a much easier and smoother process.

There are several reasons for logging:

- ▶ Error handling
- ▶ Monitoring
- ▶ Auditing

Statements like `System.out.println()` should not be used.

There are frameworks available for that. A very popular one is Apache Log4J.

Example:

```
package drawing;
import org.apache.log4j.Logger;

public class Scribble{
    final Logger logger = Logger.getLogger(Scribble.
        class);
    public Scribble(){
        logger.trace("Entering drawing mode..");
        logger.debug("Start a new drawing");
        logger.info("Connected to database");
        logger.warn("Empty input");
        logger.error("Permission denied");
        logger.fatal("Exhausted memory");
    }
}
```

The display of the messages, the format and the target (file, console, mail, database, ...) will be defined in a configuration file. This file is called `log4j2.xml` and should be placed in the classpath of the project.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="DEBUG">
    <Appenders>
        <Console name="LogToConsole" target="SYSTEM_OUT">
            <PatternLayout pattern=
                "%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
        </Console>
        <File name="LogFile" fileName="logs/app.log">
            <PatternLayout>
                <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>
            </PatternLayout>
        </File>
    </Appenders>
    <Loggers>
        <Logger name="ch.fhnw" level="debug" additivity="false">
            <AppenderRef ref="LogFile"/>
            <AppenderRef ref="LogToConsole"/>
        </Logger>
        <Logger name="org.springframework.boot" level="error"
            additivity="false">
            <AppenderRef ref="LogToConsole"/>
        </Logger>
        <Root level="error">
            <AppenderRef ref="LogFile"/>
            <AppenderRef ref="LogToConsole"/>
        </Root>
    </Loggers>
</Configuration>
```

There are many appenders available:

- ▶ ConsoleAppender
- ▶ FileAppender
- ▶ SMTPAppender
- ▶ JDBCAppender
- ▶ JMSAppender
- ▶ NTEventLogAppender
- ▶ TelnetAppender
- ▶ SyslogAppender
- ▶ SocketAppender

If a message will appear depends on the defined priority.

The following log levels are available:

- 0 TRACE Designates finer-grained informational events than the DEBUG.
- 1 DEBUG Designates fine-grained informational events that are most useful to debug an application.
- 2 INFO Designates informational messages that highlight the progress of the application at coarse-grained level.
- 3 WARN Designates potentially harmful situations.
- 4 ERROR Designates error events that might still allow the application to continue running.
- 5 FATAL Designates very severe error events that will presumably lead the application to abort.

There could be more or less levels available, depending on the used log framework. The following format options are available to set up the format of a message:

- %n newline
- %m your log message
- %p message priority (FATAL, ERROR, WARN, INFO, DEBUG, TRACE ...)
- %r millisecs since program started running
- %% percent sign in output
- %c name of your category (logger)
- %t name of current thread
- %d date and time
- %l Shortcut for %F%L%C%M
- %F Java source file name
- %L Java source line number
- %C Java class name
- %M Java method name

Brian W. Kernighan, Rob Pike, "The Practice of Programming" (1999):

*As a personal choice, we tend not to use debuggers beyond getting a stack trace or the value of a variable or two. One reason is that it is easy to get lost in details of complicated data structures and control flow; we find stepping through a program less productive than thinking harder and adding output statements and self-checking code at critical places. Clicking over statements takes longer than scanning the output of judiciously-placed displays. It takes less time to decide where to put print statements than to single-step to the critical section of code, even assuming we know where that is. More important, logging statements stay with the program; debugging sessions are transient.*

Together with the Spring Boot technology, the logging framework *Logback* is available.

Logback is intended as a successor to the popular log4j project, picking up where log4j leaves off.

There is a default configuration available which could be configured in different ways:

First, we can set our logging level within our VM Options:

- Dlogging.level.org.springframework=INFO
- Dlogging.level.ch.fhnw=TRACE

or with Maven:

```
mvn spring-boot:run  
-Dspring-boot.run.arguments=  
--logging.level.org.springframework=INFO,  
--logging.level.ch.fhnw=TRACE
```

If there is a configuration file logback.xml in the classpath, the default configuration will be overwritten by that file:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

    <property name="LOGS" value="../logs" />

    <appender name="Console"
        class="ch.qos.logback.core.ConsoleAppender">
        <layout class="ch.qos.logback.classic.PatternLayout">
            <Pattern>
                %black(%d{ISO8601}) %highlight(%-5level)
                [%blue(%t)] %yellow(%C{1.}): %msg%n%throwable
            </Pattern>
        </layout>
    </appender>

    <appender name="RollingFile"
        class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>${LOGS}/spring-boot-logger.log</file>
        <encoder
            class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
            <Pattern>%d %p %C{1.} [%t] %m%n</Pattern>
        </encoder>

        <rollingPolicy
            class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <!-- rollover daily and when the file reaches 10 MegaBytes -->
```

```
<fileNamePattern>
${LOGS}/archived/spring-boot-logger-%d{yyyy-MM-dd}.%i.log
</fileNamePattern>
<timeBasedFileNamingAndTriggeringPolicy
    class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
    <maxFileSize>10MB</maxFileSize>
</timeBasedFileNamingAndTriggeringPolicy>
</rollingPolicy>
</appender>

<!-- LOG everything at INFO level -->
<root level="info">
    <appender-ref ref="RollingFile" />
    <appender-ref ref="Console" />
</root>

<logger name="ch.fhnw" level="trace" additivity="false">
    <appender-ref ref="RollingFile" />
    <appender-ref ref="Console" />
</logger>

</configuration>
```

In a production environment, logging is normally set to the log level `warn` or higher. But sometimes it is needed, to change that level to `trace` to find out the root cause of a bug/issue.

If the log configuration is part of the software package means, a new package needs to be created. The software is exactly the same only the log level has changed.

Therefore it would make sense to have the possibility to change the log level without installing a new software package.

One way to achieve that is with the `spring-boot-starter-actuator` plugin.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</
        artifactId>
</dependency>
```

Starting with Spring Boot 2.x, most endpoints are disabled by default, so we'll also need to enable the /loggers endpoint in our application.properties file:

```
management.endpoints.web.exposure.include=loggers  
management.endpoint.loggers.enabled=true
```

Connect to the following endpoint to see the log levels of all packages:

```
http://localhost:8080/actuator/loggers
```

The following command could be used to change the log level during runtime:

```
curl -i -X POST -H 'Content-Type: application/json'  
-d '{"configuredLevel": "TRACE"}'  
http://localhost:8080/actuator/loggers/ch.fhnw
```

To find the root cause of issues it is sometimes very useful to compare text files. Especially if a running and a non running version of a software exists.

A specific editor is needed to compare two text files (e.g. Beyond Compare). On the other side, on linux based systems there are tools available for that.

The first step would be to find out if two files are identical:

```
md5 file.txt
```

```
MD5 (pom.xml) = 2fb03b258a4e117c98751b55b8af4ba5
```

The next step would be compare both files:

```
$ diff colors1.txt colors2.txt
```

```
6c6
< # indigo
_____
> # indigo-blue
11d10
< # sienna
12a12
> # darkblue
15c15
< # tomato
_____
> # tomato-red
```

You could also use the option -u (unified)

```
$ diff -u colors1.txt colors2.txt
```

```
--- colors1.txt 2005-04-01 20:45:21.954828407 +0200
+++ colors2.txt 2005-04-01 20:44:50.152742422 +0200
@@ -3,13 +3,13 @@
 # darkred
 # deeppink
 # firebrick
-# indigo
+# indigo-blue
# limegreen
# royalblue
# sandybrown
# seagreen
-# sienna
# silver
+# darkblue
```

```
# skyblue
# teal
-# tomato
+# tomato-red
```

The lines which are different are marked with - and +.

Source:

[www.brunolinux.com/02-The\\_Terminal/Diff\\_and\\_Cmp.html](http://www.brunolinux.com/02-The_Terminal/Diff_and_Cmp.html)

[www.brunolinux.com/02-](http://www.brunolinux.com/02-The_Terminal/Diff_Find_and_Md5sum.html)

[The\\_Terminal/Diff\\_Find\\_and\\_Md5sum.html](http://www.brunolinux.com/02-The_Terminal/Diff_Find_and_Md5sum.html)

Testivus-Manifest from Alberto Savoia:

- ▶ Less testing dogma, more testing karma.
- ▶ Any tests are better than no tests.
- ▶ Testing beats debugging.
- ▶ Test first, during, or after – whatever works best for you.
- ▶ If a method, technique, or tool, gives you more or better tests use it.

<http://www.developertesting.com/archives/month200702/200702>

JUnit is a unit testing framework for Java programming language. It plays a crucial role test-driven development, and is a family of unit testing frameworks collectively known as xUnit.

JUnit promotes the idea of "first testing then coding", which emphasizes on setting up the test data for a piece of code that can be tested first and then implemented. This approach is like "test a little, code a little, test a little, code a little." It increases the productivity of the programmer and the stability of program code, which in turn reduces the stress on the programmer and the time spent on debugging.

- ▶ JUnit is an open source framework, which is used for writing and running tests.
- ▶ Provides annotations to identify test methods.
- ▶ Provides assertions for testing expected results.
- ▶ Provides test runners for running tests.

- ▶ JUnit tests allow you to write codes faster, which increases quality.
- ▶ JUnit is elegantly simple. It is less complex and takes less time.
- ▶ JUnit tests can be run automatically and they check their own results and provide immediate feedback. There's no need to manually comb through a report of test results.
- ▶ JUnit tests can be organized into test suites containing test cases and even other test suites.
- ▶ JUnit shows test progress in a bar that is green if the test is running smoothly, and it turns red when a test fails.

Procedure to create a test class:

1. Create a test class for a given class A:

```
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions;  
  
public class ClassATest {  
  
    @Test  
    public void testMethodA() {  
        Assertions.assertTrue(CONDITION);  
    }  
}
```

2. Mark each test method with the annotation @Test.

3. Add assert statements to test some specific conditions.  
To test floating point numbers, the method  
`assertEquals(val1, val2, delta)` could be used.
4. Optional: Add a method and mark it with `@BeforeAll` or `@BeforeEach`.  
This method will be executed once before all tests or once before each test.
5. Optional: Add a method and mark it with `@AfterAll` or `@AfterEach`.  
This method will be executed once after all tests or once after each test.  
The test could be started within the build process (e.g. ANT, Maven) or from the IDE directly.

## With IntelliJ:

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Structure:** The project is named "karnaugh-map-simplifier". It contains a src directory with main and test packages. The test package contains CellTest, CoordinateTest, PreSumTest, and TruthTableTest.
- CoordinateTest.java:** The code for the test class is shown, including a failing test method `testGetX()`.
- Coverage:** A coverage report on the right indicates 9% classes, 4% lines covered. It lists various packages and their coverage percentages.
- Run Results:** The bottom panel shows the test results: 10 tests done, 1 failed, and 18ms execution time. The failed test is `junit.framework.AssertionFailedError: Expected 3` vs `Actual 2`.
- Bottom Status:** Shows the date and time (11:15), file encoding (UTF-8), and the current tab (UnitTests).

Figure: JUnit IntelliJ

With ANT:

```
<target name="junit" depends="compile">
    <junit printsummary="yes" haltonfailure="no">
        <!-- Project classpath , must include junit.jar
            -->
        <classpath refid="test.path" />

        <!-- test classes -->
        <classpath location="${test.classes.dir}" />

        <test name="ch.fhnw.TestMessage"
            haltonfailure="no" todir="${report.dir}">
            <formatter type="plain" />
            <formatter type="xml" />
        </test>^^I
    </junit>
</target>
```

With Maven:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.8.2</version>
    <scope>test</scope>
</dependency>
```

The tests could then be executed with

```
mvn test
```

Code coverage is a software testing metric that determines the number of lines of code that is successfully validated under a test procedure, which in turn, helps in analyzing how comprehensively a software is verified.

Developing great (bug-free) software products is the goal of any software developer. However, to accomplish this goal, developers need to ensure that the software they develop meets all the essential quality characteristics, **correctness, reliability, effectiveness, security, and maintainability**.

Code coverage is one such software testing metric that can help in assessing the test performance and quality aspects of any software.

For developers, this metric can help in dead code detection and elimination. On the other hand, it can help to check missed or uncovered test cases.

A coverage report could be created in different ways with different tools.

With IntelliJ:

A coverage report in IntelliJ could be created with the **Run Tests with Coverage** feature.

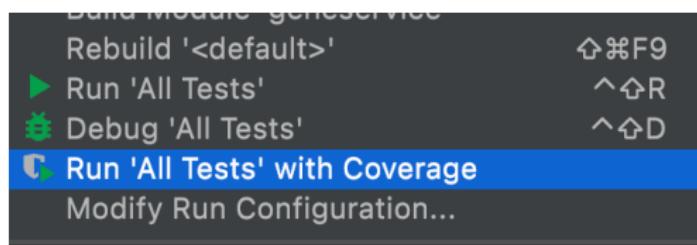


Figure: Coverage IntelliJ

A detailed report will be created which gives feedback on how many classes, methods and lines of code are covered by a unit test.

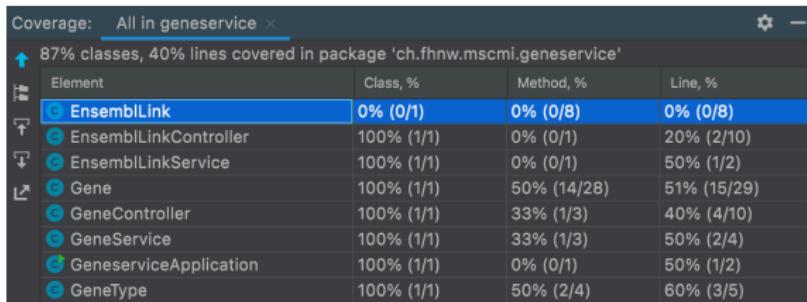


Figure: Coverage IntelliJ

With Maven:

Jacoco is an open source project, which can be used to create a coverage report. It integrates with IDEs (e.g. Eclipse), build tools (e.g. Maven, Gradle) or continuous integration systems (e.g. Jenkins).

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.8</version>
  <executions>
    <execution>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
```

```
<id>report</id>
<phase>prepare-package</phase>
<goals>
    <goal>report</goal>
</goals>
</execution>
</executions>
</plugin>
```

If the command mvn package is executed (includes tests), a report will be generated in target/site/jacoco:

## ch.fhnw.mscmi.geneservice

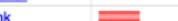
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cqty	Missed	Lines	Missed	Methods	Missed	Classes
Gene		44%	n/a	14	29	28	43	14	29	0	1	
EnsemblLinkController		15%		0%	2	3	8	10	1	2	0	1
GeneController		38%	n/a	2	4	6	10	2	4	0	1	
EnsemblLink		0%	n/a	9	9	13	13	9	9	1	1	
GeneService		50%	n/a	2	4	2	4	2	4	0	1	
GeneType		52%	n/a	2	5	4	7	2	5	0	1	
EnsemblLinkService		37%	n/a	1	2	1	2	1	2	0	1	
GeneserviceApplication		37%	n/a	1	2	2	3	1	2	0	1	
Total	189 of 289	34%	2 of 2	0%	33	58	64	92	32	57	1	8

Figure: Coverage JaCoCo

*Always maintain constantly shippable code. It is incredibly important that your software can always be run by your team.*

*Don't break the nightly build!* is a cardinal rule in software development shops that post a freshly built daily product version every morning for their testers.

Continuous Integration is a software development practice in which developers are required to frequently commit changes to the source code in a shared repository. Each commit is then continuously pulled and built.

The goal of Continuous Integration is to provide rapid feedback so that if a defect is introduced into the code base, it can be identified and corrected as soon as possible.

That approach will ensure that:

- ▶ identify defects as soon as possible
- ▶ all sources are in the repository to build the system
- ▶ all available tests are executed successfully

Important are frequent commit and push operations. Working many days in a row without pushing the results into the repository will increase the probability that something will fail.

A CI server ensures, that new code in the repository will directly be included in the build process.

It is also possible to sent out mail notifications in case of a successful build.

This process is also called *Smoke Testing*. Smoke Testing is a software testing process that determines whether the deployed software build is stable or not. Smoke testing is a confirmation for QA team to proceed with further software testing. It consists of a minimal set of tests run on each build to test software functionalities.

Available continuous integration systems:

- ▶ CruiseControl [cruisecontrol.sourceforge.net](http://cruisecontrol.sourceforge.net)
- ▶ Jenkins/Hudson [jenkins-ci.org](http://jenkins-ci.org)
- ▶ Bamboo [www.atlassian.com/software/bamboo](http://www.atlassian.com/software/bamboo)
- ▶ Continuum [continuum.apache.org](http://continuum.apache.org)
- ▶ TeamCity [www.jetbrains.com/teamcity](http://www.jetbrains.com/teamcity)
- ▶ GitLab [www.gitlab.com](http://www.gitlab.com)

Jenkins offers a simple way to set up a continuous integration or continuous delivery (CI/CD) environment for almost any combination of languages and source code repositories using pipelines, as well as automating other routine development tasks. While Jenkins doesn't eliminate the need to create scripts for individual steps, it does give you a faster and more robust way to integrate your entire chain of build, test, and deployment tools than you can easily build yourself.

Jenkins will be installed as a server component. You could deploy it as a web application (e.g. Tomcat, JBoss) or run it with the integrated server.

The configuration of the system and the build processes are set up by using the web front end application.

The screenshot shows the Jenkins web interface. On the left is a sidebar with links: New Job, Manage Jenkins, People, Build History, Redmine, and My Views. Below these are sections for Build Queue (empty) and Build Executor Status (two idle executors). The main area is titled "Continuous Integration Build Server" and displays a table of active projects. The table has columns: S (Status), W (Workload), Job (name), Last Success, Last Failure, and Last Duration. Projects listed are Project B, Project A, and Project C, each with a green circular icon and a yellow sun icon.

S	W	Job	Last Success	Last Failure	Last Duration
		Project B	1 hr (#26)	2 hr (#24)	18 sec
		Project A	1 mo 5 days (#31)	3 hr (#25)	4.2 sec
		Project C	28 days (#32)	3 mo 0 days (#23)	3.5 sec

Icon:    Legend: for all for failures for just latest builds

Page generated: Jun 22, 2011 9:04:00 AM Jenkins ver. 1.399

Figure: Jenkins GUI (Source: J. Ferguson Smart)

In addition, there is a REST interface and a command line tool available.

Jenkins unterscheidet die folgenden Projekt-Typen:

- ▶ Free-Style-Projekt: ausführen von Funktionen, die in Shell-, Windows Batch- oder Ant-Skripts enthalten sind,
- ▶ Maven-2-Projekt: Durchführung eines Maven-Build-Prozesses,
- ▶ Multikonfigurationsprojekt: mehrfache Durchführung eines Builds für unterschiedliche Konfigurationen (Matrix-Build).

Die Auslösung eines Build-Vorganges kann auf verschiedene Weise geschehen:

- ▶ nachdem ein Build-Vorgang beendet ist,
- ▶ in periodischen Intervallen: MIN HOUR DOM MONTH DOW
- ▶ bei Änderungen im SCM
- ▶ explizit

GitLab CI/CD is the part of GitLab that could be used for all of the continuous methods (Continuous Integration, Delivery, and Deployment). With GitLab CI/CD, it is possible to test, build, and publish a software project with no third-party application or integration needed.

GitLab CI/CD fits in a common development workflow.

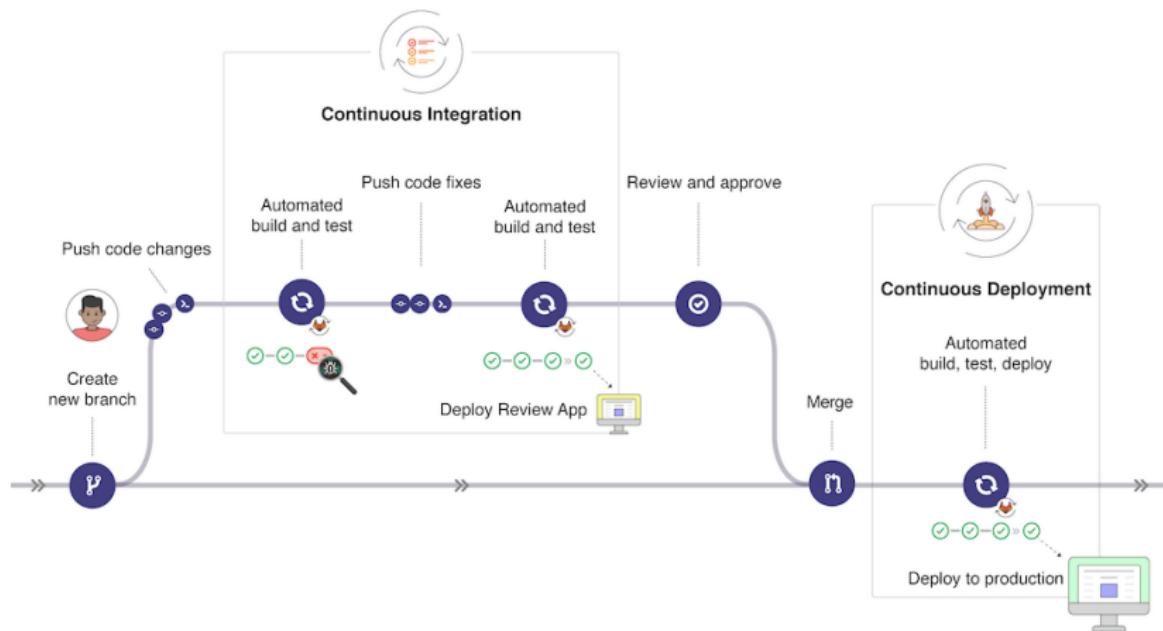


Figure: Gitlab CI Workflow

A file `.gitlab-ci.yml` needs to be added to the project. The `.gitlab-ci.yml` file is a YAML file where specific instructions for GitLab CI/CD will be configured.

This file contains:

- ▶ The structure and order of jobs that the runner should execute.
- ▶ The decisions the runner should make when specific conditions are encountered.

Sample `.gitlab-ci.yml` file:

```
build-job:  
  stage: build  
  script:  
    - echo "Hello, $GITLAB_USER_LOGIN!"  
  
test-job1:  
  stage: test  
  script:  
    - echo "This job tests something"  
  
test-job2:  
  stage: test  
  script:  
    - echo "This job tests something, but takes more time"  
    - sleep 20  
  
deploy-prod:
```

```
stage: deploy
script:
  - echo "This job deploys something from the $CI_COMM
```

Make your code ready for a production deployment!

1. Extend your project with one unit test (e.g. search by id).
  - ▶ Run the unit test from the IDE
  - ▶ Run the unit test with Maven and create a report
2. Use Checkstyle and create a report (Maven or IDE)
3. Improve your application with a log file. Create some useful log outputs on different levels
  - ▶ Use the actuator plugin to change the log level during runtime
4. Check the application with JMeter. Create ~1000 users and call `byid` for each user. Report the memory usage before the call and after the call.
5. Add useful comments and create a documentation of your source code
6. Check in your source code and create a tag

**Quality** Degree to which a set of inherent characteristics fulfills requirements (EN ISO 9000:2005)

**Quality Management** the act of overseeing different activities and tasks within an organization to ensure that products and services offered, as well as the means used to provide them, are consistent. It helps to achieve and maintain a desired level of quality within the organization.

**Quality Assurance** act or process of confirming that the quality requirements are being met.

Quality is

- ▶ transcendent - universally recognizable but not precisely defined (innate excellence).
- ▶ product-based - a precise and measurable variable.
- ▶ manufacturing-based - conformance to production and engineering requirements.
- ▶ user-based - “fitness for intended use.”
- ▶ value-based - costs and benefit.

Usability	Effectivity	Completeness
		Correctness
		Understandability
	Efficiency	Resources
		Performance
		Interoperability
	Reliability	Robustness
		Security
		Recovery
		Availability
		Controllability
		Verifiability

Maintainability	Modifiability	Transparency Modularity Conforming to Standards
	Adaptability	Portability Universality Connectability

Coding conventions are a set of guidelines for a specific programming language that recommend programming style, practices, and methods for each aspect of a program written in that language. These conventions usually cover file organization, indentation, comments, declarations, statements, white space, naming conventions, programming practices, programming principles, programming rules of thumb, architectural best practices, etc. These are guidelines for software structural quality. Software programmers are highly recommended to follow these guidelines to help improve the readability of their source code and make software maintenance easier. Coding conventions are only applicable to the human maintainers and peer reviewers of a software project. Conventions may be formalized in a documented set of rules that an entire team or company follows, or may be as

informal as the habitual coding practices of an individual. Coding conventions are not enforced by compilers.

Code conventions are important to programmers for a number of reasons:

- ▶ 40%-80% of the lifetime cost of a piece of software goes to maintenance.
- ▶ Hardly any software is maintained for its whole life by the original author.
- ▶ Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- ▶ If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

## Criterias:

- ▶ Conciseness: simple, clear, understandable instructions and processes.
- ▶ Compliancy: uniform naming for variables, methods, classes, packages, ...
- ▶ Modularity: clear encapsulated packages, classes and methods

## Examples

- ▶ Google [google.github.io/styleguide/javaguide.html](https://google.github.io/styleguide/javaguide.html)
- ▶ Sun [www.oracle.com/technetwork/java/index-135089.html](http://www.oracle.com/technetwork/java/index-135089.html)

### 1. File Organization

- 1.1 Files longer than 2000 lines are cumbersome and should be avoided.
- 1.2 Each Java source file contains a single public class or interface. When private classes and interfaces are associated with a public class, you can put them in the same source file as the public class. The public class should be the first class or interface in the file.
- 1.3 Java source files have the following ordering:
  - ▶ Beginning comments
  - ▶ Package and Import statements
  - ▶ Class and interface declarations

- 1.4 The first component of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981.

## 2. Declarations

- 2.1 One declaration per line is recommended since it encourages commenting. In other words,

```
int level; // indentation level  
int size; // size of table
```

is preferred over

```
int level, size;
```

- 2.2 When coding Java classes and interfaces, the following formatting rules should be followed:

- ▶ No space between a method name and the parenthesis “(“ starting its parameter list
- ▶ Open brace “{” appears at the end of the same line as the declaration statement
- ▶ Closing brace “}” starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the “}” should appear immediately after the “{”

- ▶ Methods are separated by a blank line.

```
class Sample extends Object {  
    int ivar1;  
    int ivar2;  
  
    Sample(int i, int j) {  
        ivar1 = i;  
        ivar2 = j;  
    }  
  
    int emptyMethod() {}  
  
    ...  
}
```

## 3. Naming Conventions

- 3.1 Class and Interface names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).
- 3.2 Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.
- 3.3 Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore \_ or dollar sign \$ characters, even though both are allowed.  
Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use.

One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.

- 3.4 The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("\_"). (ANSI constants should be avoided, for ease of debugging.)

## 4. Programming Practices

- 4.1 Don't make any instance or class variable public without good reason. Often, instance variables don't need to be explicitly set or gotten-often that happens as a side effect of method calls. One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behavior. In other words, if you would have used a struct instead of a class (if Java supported struct), then it's appropriate to make the class's instance variables public.
- 4.2 It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others-you shouldn't assume that other programmers know precedence as well as you do.

```
if (a == b && c == d)      // AVOID!  
if ((a == b) && (c == d)) // RIGHT
```

HtmlUnit is a UI-Less browser for Java programs. It models HTML documents and provides an API that allows you to invoke pages, fill out forms, click links, etc... just like you do in your normal browser. It is typically used for testing purposes or to retrieve information from web sites.

HtmlUnit is not a generic unit testing framework. It is specifically a way to simulate a browser for testing purposes and is intended to be used within another testing framework such as JUnit

Maven Dependency:

```
<dependency>
    <groupId>net.sourceforge.htmlunit</groupId>
    <artifactId>htmlunit</artifactId>
    <version>2.61.0</version>
</dependency>
```

Selenium is a free (open-source) automated testing framework used to validate web applications across different browsers and platforms. Selenium could be used with multiple programming languages like Java, C#, Python etc to create Selenium Test Scripts. Testing done using the Selenium testing tool is usually referred to as Selenium Testing.

Selenium Software is not just a single tool but a suite of software, each piece catering to different Selenium QA testing needs of an organization. Here is the list of tools

- ▶ Selenium Integrated Development Environment (IDE)
- ▶ Selenium Remote Control (RC)
- ▶ WebDriver
- ▶ Selenium Grid

Selenium WebDriver is a web framework that permits you to execute cross-browser tests. This tool is used for automating web-based application testing to verify that it performs expectedly. Selenium WebDriver allows you to choose a programming language to create test scripts.

WebDriver contains the following 4 components:

- ▶ Selenium Client library
- ▶ JSON wire protocol
- ▶ Browser Drivers
- ▶ Browsers

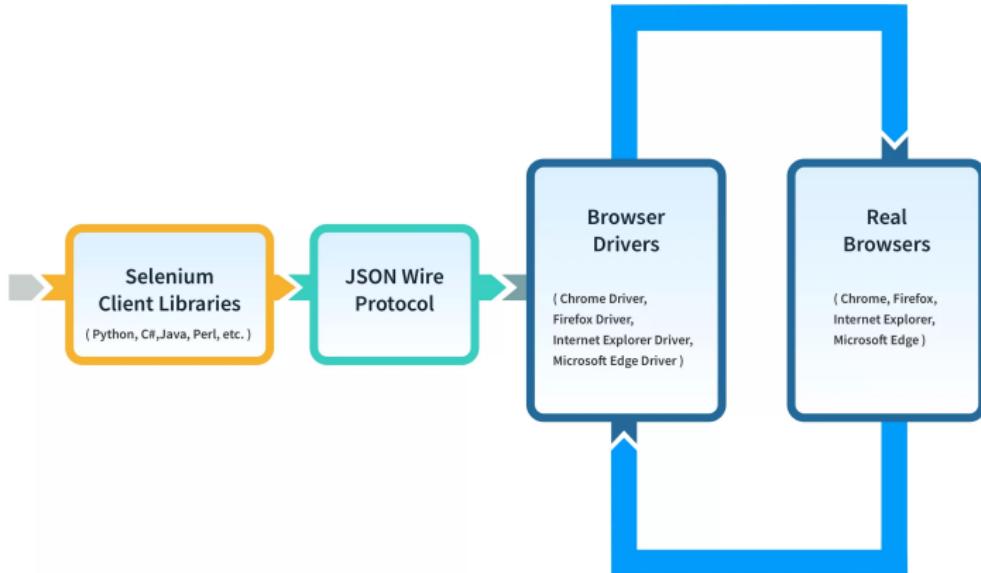


Figure: Selenium WebDriver (Source: [browserstack.com](https://www.browserstack.com))

Here are the basic steps which needs to be performed when writing a Selenium test script:

1. Create a WebDriver instance.
2. Navigate to a webpage.
3. Locate a web element on the webpage via locators in selenium.
4. Perform one or more user actions on the element.
5. Preload the expected output/browser response to the action.
6. Run test.
7. Record results and compare results from them to the expected output.

A selenium test script in the Java language could look like the following:

```
System.setProperty("webdriver.chrome.driver",  
    ".../chromedriver");  
WebDriver driver = new ChromeDriver();  
driver.get(URL);  
  
String pageTitle = driver.getTitle();  
WebElement soDropdown = driver.findElement(By.id(  
    "inputform:searchoption_input"));
```

*Designing documentation to be consulted, not read!*

A software project could contain several ways of documentation:

- ▶ Requirements (e.g. user stories, use cases)
- ▶ System architecture, design document
- ▶ Code documentation
- ▶ Operation guidelines
- ▶ User guide
- ▶ API description
- ▶ ... and many more

The first contact to a software project is in many cases the source repository. Each repository should contain a README file which contains the most important information about the project.

- ▶ General project information
- ▶ Interaction with other systems?
- ▶ Which credentials are used?
- ▶ How to build?
- ▶ How to execute the tests?

The format which is used for the README file is not defined. Mostly it is written in plain text or with the Markdown language.

For Maven projects, a lot of information about a project could also be retrieved from the `pom.xml` file.

- ▶ Dependencies
- ▶ Available profiles
- ▶ Version
- ▶ ... and many more

Javadoc is a documentation generator for the Java language for generating API documentation in HTML format from Java source code. The HTML format is used for adding the convenience of being able to hyperlink related documents together.

The `doc comments` format used by Javadoc is the de facto industry standard for documenting Java classes. Some IDEs (e.g. IntelliJ IDEA, NetBeans and Eclipse) automatically generate Javadoc HTML. Many file editors assist the user in producing Javadoc source and use the Javadoc info as internal references for the programmer.

Javadoc does not affect performance in Java as all comments are removed at compilation time. Writing comments and Javadoc is for better understanding the code and thus better maintaining it.

With Maven (included in the report):

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <version>3.4.0</version>
    </plugin>
  </plugins>
</reporting>
```

```
mvn site
```

With Maven (standalone):

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <version>3.4.0</version>
    </plugin>
  </plugins>
</build>
```

```
mvn javadoc:javadoc
```



Figure: Security Issues in the healthcare environment

Software security should always be a high priority for product developers and users. If you don't prioritize security, you and your customers will inevitably suffer losses from malicious attacks. The aim of the attacks may be to steal data or hijack a computer for some criminal purpose. Some attacks try to extort money from a user by encrypting data and demanding a fee for the decryption key, or by threatening a denial of service attack on their servers.

There are the following type of threats:

- ▶ **Confidentiality**

Protect data from un-authorized access

- ▶ **Integrity**

Protect data from being deleted or modified

- ▶ **Availability**

Keep the system (and available data) up and running

Also called the CIA triad.

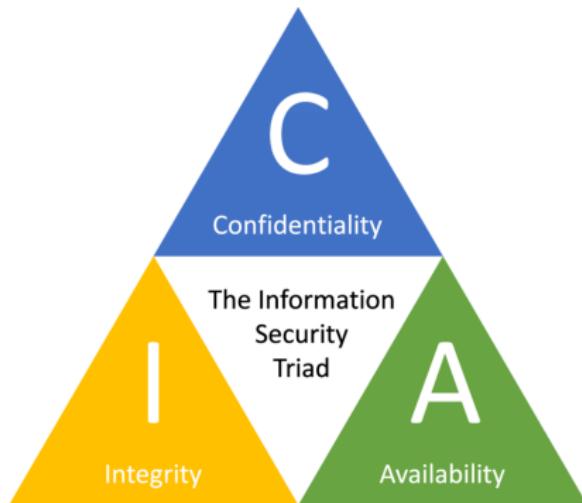


Figure: CIA Triad (Source: researchgate.net)

Depending on the software system and the underlying infrastructure the following topics needs to be considered:

- ▶ Authentication/Authorization
- ▶ Backup
- ▶ Attack monitoring
- ▶ System infrastructure management
- ▶ Data encryption

Many types of attack may affect a software system. They depend on the type of system, the way it has been implemented, the potential vulnerabilities in the system, and the environment where the system is used. The most possible attacks are:

- ▶ Injection attacks
- ▶ Cross-site scripting attacks
- ▶ Session hijacking attacks
- ▶ DoS (denial of service) attacks
- ▶ Bruteforce attacks

Authentication is the process of identifying users and validating who they claim to be. One of the most common and obvious factors to authenticate identity is a password. If the user name matches the password credential, it means the identity is valid, and the system grants access to the user.

## Examples

- ▶ Password-based authentication
- ▶ Passwordless authentication
- ▶ Two factor / Multi factor
- ▶ Single sign on
- ▶ Social authentication

Authorization happens after a user's identity has been successfully authenticated. It is about offering full or partial access rights to resources like database, funds, and other critical information to get the job done.

## Examples

- ▶ Role-based access controls
- ▶ JSON web token
- ▶ SAML
- ▶ OpenID authorization
- ▶ OAuth

Encryption is the process of making a document unreadable by applying an algorithmic transformation to it. The encryption algorithm uses a secret key as the basis of this transformation.

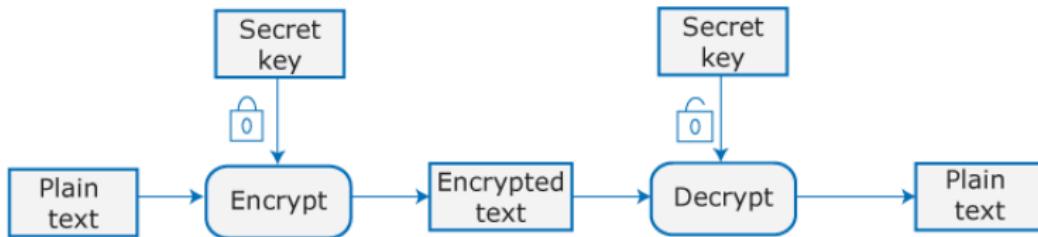


Figure: Encryption process (Source: Sommerville, Software Engineering)

- ▶ **Symmetric Encryption**

Uses the same key for encryption and decryption

- ▶ **Asymmetric Encryption**

Uses a different key for encryption and decryption

Encryption could be used to reduce the damage that may occur from data theft. If information is encrypted, it is impossible, or very expensive, for thieves to access and use the unencrypted data.

The practicality of encryption depends on the encryption context:

- ▶ **in transit**

Data is moved from one computer to another

- ▶ **at rest**

Data is stored

- ▶ **in use**

Data is actively processed

The most important differences between symmetric and asymmetric encryption are:

- ▶ Symmetric encryption uses a single key that needs to be shared among the people who need to receive the message while asymmetric encryption uses a pair of public key and a private key to encrypt and decrypt messages when communicating.
- ▶ Asymmetric encryption was introduced to complement the inherent problem of the need to share the key in symmetric encryption model, eliminating the need to share the key by using a pair of public-private keys.
- ▶ Asymmetric encryption takes relatively more time than the symmetric encryption.

Vigenere Cipher is a method of encrypting alphabetic text. It uses a simple form of polyalphabetic substitution. A polyalphabetic cipher is any cipher based on substitution, using multiple substitution alphabets. The encryption of the original text is done using the Vigenere table.

		Plaintext																									
		A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Key	A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
	B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
	C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
	D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
	E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
	F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
	G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
	H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
	I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
	J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
	K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
	L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
	M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
	N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
	O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
	P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
	R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
	S	S	T	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	
	T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
	U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
	V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
	W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
	X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
	Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
	Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Figure: Vigenere Table (Source: javatpoint.com)

Original Text: HELLO

Key: KEY

Encrypted Text: RIJVM

RSA (Rivest–Shamir–Adleman) is a public-key cryptosystem that is widely used for secure data transmission.

In a public-key cryptosystem, the encryption key is public and distinct from the decryption key, which is kept secret (private). An RSA user creates and publishes a public key based on two large prime numbers, along with an auxiliary value. The prime numbers are kept secret. Messages can be encrypted by anyone, via the public key, but can only be decoded by someone who knows the prime numbers (or the private key).

The security of RSA relies on the practical difficulty of factoring the product of two large prime numbers.

The General Data Protection Regulation (GDPR) is the toughest privacy and security law in the world. Though it was drafted and passed by the European Union (EU), it imposes obligations onto organizations anywhere, so long as they target or collect data related to people in the EU. The regulation was put into effect on May 25, 2018. The GDPR will levy harsh fines against those who violate its privacy and security standards, with penalties reaching into the tens of millions of euros.

Source: [gdpr.eu](http://gdpr.eu)

The GDPR applies in principle to all automated personal data processing and in some cases also manual processing of personal data. Personal data is any information that refers to an identified or identifiable natural person.

The GDPR applies to personal data processing linked to the EU, either when the entity processing the personal data is established within the EU or when an entity outside the EU offers goods and services to people within the Union or monitors their behaviour here.

The GDPR defines the following legal terms:

▶ **Personal data**

Personal data is any information that relates to an individual who can be directly or indirectly identified. Names and email addresses are obviously personal data. Location information, ethnicity, gender, biometric data, religious beliefs, web cookies, and political opinions can also be personal data. Pseudonymous data can also fall under the definition if it's relatively easy to ID someone from it.

▶ **Data processing**

Any action performed on data, whether automated or manual.

▶ **Data subject**

The person whose data is processed (e.g. customers, site visitors).

▶ **Data controller**

The person who decides why and how personal data will be processed (e.g. owner, employee).

▶ **Data processor**

A third party that processes personal data on behalf of a data controller.

If data is processed, the following principles needs to be considered:

▶ **Lawfulness, fairness and transparency**

Organisations need to ensure their data collection practices don't break the law and that they aren't hiding anything from data subjects.

▶ **Purpose limitation**

Organisations should only collect personal data for a specific purpose, clearly state what that purpose is, and only collect data for as long as necessary to complete that purpose.

▶ **Data minimization**

Organisations must only process the personal data that they need to achieve its processing purposes.

## ► Accuracy

The accuracy of personal data is integral to data protection.

The GDPR states that *every reasonable step must be taken* to erase or rectify data that is inaccurate or incomplete.

## ► Storage limitation

Organisations need to delete personal data when it's no longer necessary.

## ► Integrity and confidentiality

Processing must be done in such a way as to ensure appropriate security, integrity, and confidentiality (e.g. by using encryption).

## ► Accountability

The data controller is responsible for being able to demonstrate GDPR compliance with all of these principles.

In the following exercise you have created a system to monitor patients with a smart phone. The application will upload personal data to the cloud. The data will be encrypted on the device before the upload. For the data encryption process, you decided to use RSA encryption.

Before the data could be used for further processing, it needs to be decrypted.

Perform the following steps:

1. Create a key pair

- ▶ Use the available Java program

<https://gitlab.fhnw.ch/david.herzig/dataencryption>

- ▶ Use openssl

```
openssl genrsa -out private_key.pem 2048
openssl pkcs8 -topk8 -inform PEM -outform
    DER -in private_key.pem -out private_key
        .der -nocrypt
openssl rsa -in private_key.pem -pubout -
    outform DER -out public_key.der
```

2. Publish the public key

3. Convert an encrypted file

4. Do you see any potential risks with that approach?

What is the difference between software and medical software?

Medical software is any software item or system used within a medical context. Examples are:

- ▶ standalone software used for diagnostic or therapeutic purposes
- ▶ software embedded in a medical device
- ▶ software that drives a medical device or determines how it is used
- ▶ software that acts as an accessory to a medical device
- ▶ software used in the design, production, and testing of a medical device
- ▶ software that provides quality control management of a medical device

A software which is used to control a x-ray machine is a medical device software.

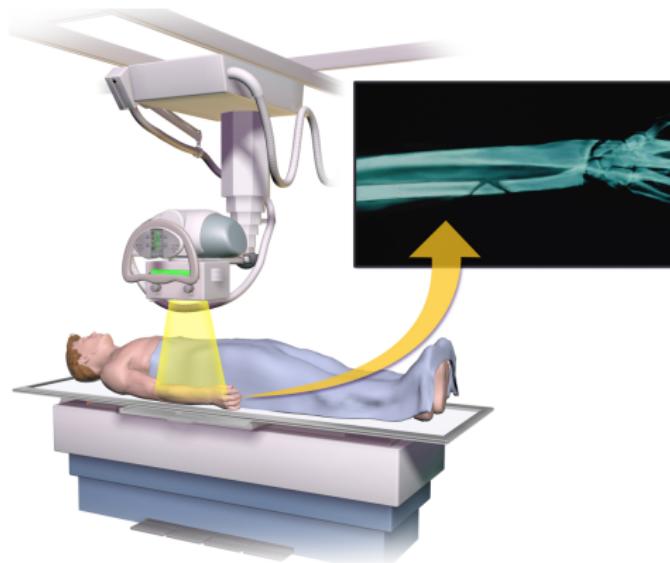


Figure: X-Ray machine (Source: Wikimedia Commons)

Mission of the FDA ([www.fda.gov](http://www.fda.gov)):

*The Food and Drug Administration is responsible for protecting the public health by ensuring the **safety**, **efficacy**, and **security** of human and veterinary drugs, biological products, and medical devices; and by ensuring the safety of our nation's food supply, cosmetics, and products that emit radiation.*

- ▶ Efficacy
- ▶ Safety - Usability
- ▶ Security - Cybersecurity

Not all software is subject to FDA regulation!

## History of the FDA

- ▶ Food and Drug Act 1906 - poisonous food additives
- ▶ Food, Drug and Cosmetics Act 1938
  - ▶ Elixir Sulfanilamide (antibiotic) killed over 100 people
  - ▶ Expands FDA scope to cosmetics and therapeutics devices
  - ▶ Authorized factory inspections

- ▶ 1962 Thalomide disaster
  - ▶ 10000 children were born with thalomide-related disabilities worldwide
  - ▶ Congress requires drugs to get approval from FDA prior to release to market

- ▶ 1971 Dalkon shield - intrauterine contraceptive device
  - ▶ 1974 approx. 2.5 million women
  - ▶ Pelvic inflammatory disease. Damage to reproductive organs, sepsis, infertility or sterility, miscarriage, even death
  - ▶ Medical Device Amendment required premarket approval for medical devices

- ▶ 1982 Therac-25 - radiation therapy
  - ▶ Three modes: Electron, X-Ray, Field-Light position
  - ▶ X-Ray 100x more energy, usage of a flattener
  - ▶ Therac-25 uses a software to ensure safety
  - ▶ 1985 - 1987 6 accidents. Massive overdose of radiation
  - ▶ Software quality issues: Safety is a quality of the system in which the software was used; it is not a quality of the software itself.

Law is reactive!

Definition (FDA):

*Federal Food, Drug, and Cosmetic Act (FD&C Act)*

*Instrument, apparatus, implement, machine, contrivance, implant, in vitro reagent, or other similar or related article, including a component part, or accessory which is:*

- ▶ Recognized in the official National Formulary, or the United States Pharmacopoeia, or any supplement to them
- ▶ Intended for use in the diagnosis of disease or other conditions, or in the cure, mitigation, treatment, or prevention of disease, in man or other animals, or
- ▶ Intended to affect the structure or any function of the body of man or other animals, and which does not achieve its primary intended purposes through chemical action within or on the body of man or other animals and which is not dependent upon being metabolized for the achievement of its primary intended purposes.

## Definition (FDA):

*Software intended to be used for one or more medical purposes that perform these purposes without being part of a hardware medical device.*

Use of Software as a Medical Device is continuing to increase. It can be used across a broad range of technology platforms, including medical device platforms, commercial *off-the-shelf* platforms, and virtual networks, to name a few. Such software was previously referred to by industry, international regulators, and health care providers as *standalone software*, *medical device software*, and/or *health software*, and can sometimes be confused with other types of software.

## SaMD examples:

- ▶ Diagnosis, prevention, monitoring, treatment or alleviation of disease
  - ▶ SaMD that performs analysis of clinical samples that help with disease diagnosis.
  - ▶ SaMD that helps in monitoring sleep apnea by using the microphone of a smart device to detect breathing pattern during sleep and sounds a tone to rouse the sleeper when detecting interrupted breathing.
  - ▶ SaMD that uses data from individuals for predicting risk score for developing stroke or heart disease for creating prevention or interventional strategies.
- ▶ Disease management
  - ▶ SaMD that can provide information by taking pictures, monitoring the growth or to supplement other information for a healthcare provider for the uses of disease monitoring.

- ▶ Control of conception
- ▶ In-vitro diagnostics and sterilization

Not SaMD examples:

- ▶ A software that is embedded as part of a hardware medical device and is necessary to drive the intended medical purpose
- ▶ Software to serve as electronic patient records
- ▶ Software for administrative support of health care facility
- ▶ Software to maintain a healthy lifestyle (apps from play store (Android) and/or app store (iPhone))

Be aware that a software which is not classified as a medical device today may become a medical device tomorrow!

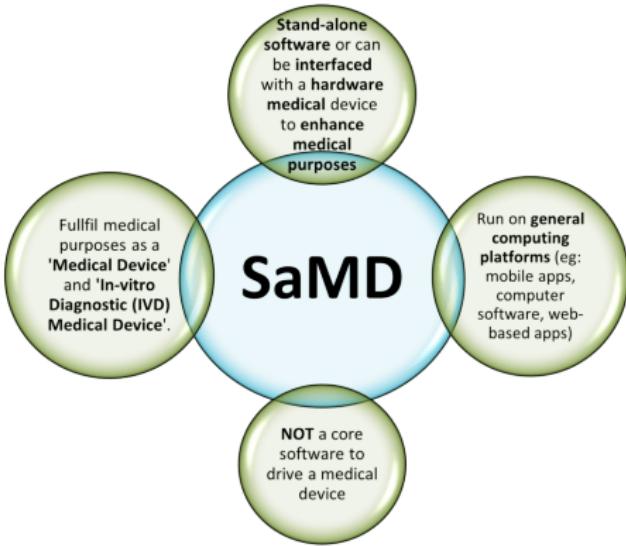


Figure: General requirements to be classified as a SaMD (Source: [www.kvalito.ch](http://www.kvalito.ch))

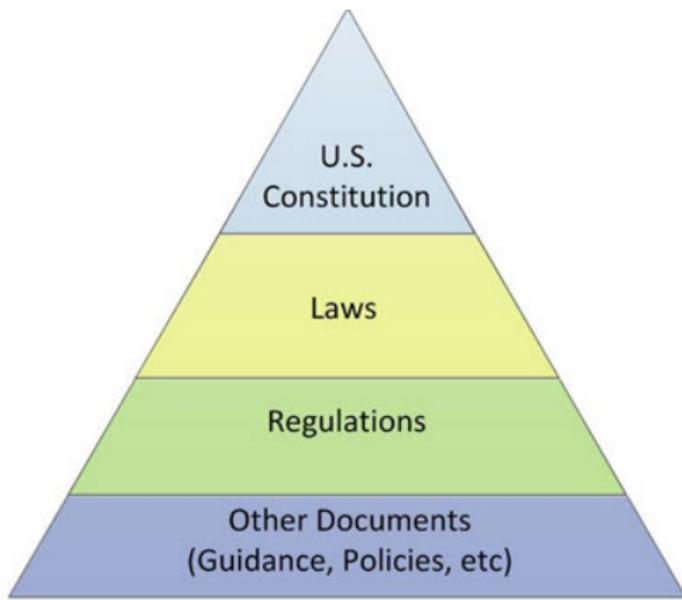


Figure: U.S. Law (Source: [www.phe.gov](http://www.phe.gov))

## ▶ Laws

Laws are passed by both houses of the U.S. Congress and then signed by the president.

## ▶ Regulations

Agencies are tasked with implementing laws by drafting regulation. They take the broad ideas of the bill and fill in the details of what needs to be done and how it will be enforced.

## ▶ Guidance

Guidance is supplemental material published by an agency that helps clarify existing rules. These include interagency statements, advisories, bulletins, policy statements, questions and answers and frequently asked questions. It is not subject to rule-making procedures, so there is no proposal or comment period. Guidance can be helpful, but it is not binding.

- ▶ Regulation QSR (Quality System Regulation, US)
- ▶ Guidance GPSV (General Principles of Software Validation, US)
- ▶ Regulation: MDR (Medical Device Regulation, EU)
- ▶ Guidance: MDCG (Guidance of Cybersecurity for medical devices, EU)

## Definition:

*A set of criteria within an industry relating to the standard functioning and carrying out of operations in their respective fields of production. In other words, it is the generally accepted requirements followed by the members of an industry.*

1. **ISO 13485:** Medical Devices - Quality Management Systems
2. **ISO 14971:** Application of Risk Management to medical devices
3. **IEC 62304:** Medical device software - system lifecycle processes
4. **IEC 62366:** Medical devices - Application of usability engineering to medical devices

If a software product is classified as medical device, software engineers must pay attention to different regulations.

The decision on whether a software should be classified as a medical device is up to the manufacturer themselves.

There are several sources available to give guidance for the classification. This guidance is not *legally binding*.

## IMDRF (International Medical Device Regulators Forum)

*Software as a Medical Device: Possible Framework for Risk Categorization and Corresponding Considerations*

State of Healthcare situation or condition	Significance of information provided by SaMD to healthcare decision		
	Treat or diagnose	Drive clinical management	Inform clinical management
Critical	IV	III	II
Serious	III	II	I
Non-serious	II	I	I

Figure: Classification (Source: IMDRF)

		Significance of Information provided by the MDSW to a healthcare situation related to diagnosis/therapy		
		High Treat or diagnose ~ IMDRF 5.1.1	Medium Drives clinical management ~ IMDRF 5.1.2	Low Informs clinical management (everything else)
State of Healthcare situation or patient condition	Critical situation or patient condition ~ IMDRF 5.2.1	Class III <i>Category IV.i</i>	Class IIb <i>Category III.ii</i>	Class IIa <i>Category II.i</i>
	Serious situation or patient condition ~ IMDRF 5.2.2	Class IIb <i>Category III.ii</i>	Class IIa <i>Category II.ii</i>	Class IIa <i>Category I.ii</i>
	Non-serious situation or patient condition (everything else)	Class IIa <i>Category II.iii</i>	Class IIa <i>Category I.iii</i>	Class IIa <i>Category I.i</i>

Figure: Classification (Source: IMDRF)

- MDR Classification Rule 11 for Medical Device Software

How it does **NOT** work:

1. Implement software
2. Perform validation study
3. Submit results to FDA
4. Approval

## How it works:

1. Establish quality process for software creation
2. Design software and document
3. Perform risk analysis
4. Review process with FDA (optional)
5. Implement software and document
6. Perform validation study
7. Submit entire process to FDA
8. Approval/Clearance
9. Post-market monitoring

To get approval/clearance for a SaMD product, the regulation body (e.g. FDA) will not test your code/software. The regulation body will review/check if there is a process in place and there is a proof that the process was followed correctly.

*The quality system regulation includes requirements related to the methods used in, and the facilities and controls used for, designing, manufacturing, packaging, labeling, storing, installing, and servicing of medical devices intended for human use.*

FDA Quality System Regulation  
MDR (EU) 2017/745

- ▶ **Verification**

specified requirements have been fulfilled

- ▶ **Validation**

particular requirements for a specific intended use can be consistently fulfilled

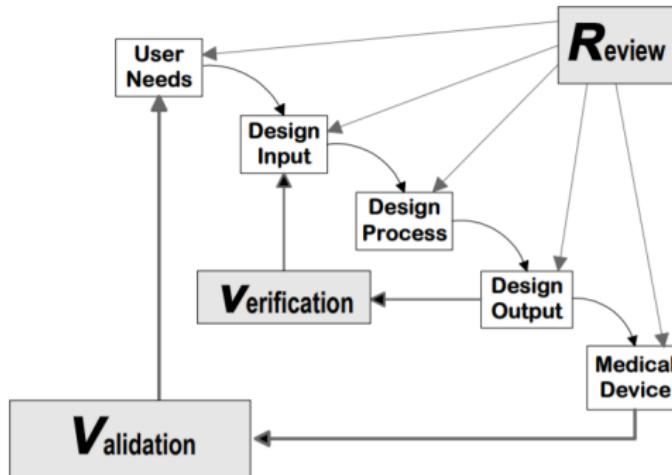


Figure: Verification vs Validation (Source: ris.world)

## ► **Design input**

means the physical and performance requirements of a device that are used as a basis for device design.

IEC 62304: The interpretation of customer needs into formally documented medical device requirements.

## ► **Specification**

means any requirement with which a product, process service, or other activity must conform.

IEC 62304: ...are the formally documented specifications of what the software does to meet the customer needs and the design inputs.

## ► **Design Output**

means the result of a design effort at each design phase and at the end of the total design effort

▶ **Design review**

means a documented, comprehensive, systematic examination of a design to evaluate the adequacy of the design requirements, to evaluate the capability of the design to meet these requirements, and to identify problems.

▶ **Design history file**

means a compilation of records which describes the design history of a finished device.

*The implication is that in a formal design process we need formal (signed, dated and properly archived) documents that verify that the process was followed.*

## General Principles of Software Validation (GSPV)

This guidance outlines general validation principles that the FDA considers to be applicable to the validation of medical device software or the validation of software used to design, develop, or manufacture medical devices.

1. Purpose
2. Scope
3. Context for Software Validation
4. Principles of Software Validation
5. Activities and Tasks
6. Validation of Automated Process Equipment

## Scope

The scope of this guidance is somewhat broader than the scope of validation in the strictest definition of that term. Planning, verification, testing, traceability, configuration management, and many other aspects of good software engineering discussed in this guidance are important activities that together help to support a final conclusion that software is validated.

How should the software be developed?

This guidance recommends an integration of software life cycle management and risk management activities. Based on the intended use and the safety risk associated with the software to be developed, the software developer should determine the specific approach, the combination of techniques to be used, and the level of effort to be applied. While this guidance does not recommend any specific life cycle model or any specific technique or method, it does recommend that software validation and verification activities be conducted throughout the entire software life cycle.

## Software updates

The FDA's analysis of 3140 medical device recalls conducted between 1992 and 1998 reveals that 242 of them (7.7%) are attributable to software failures. Of those software related recalls, 192 (or 79%) were caused by software defects that were introduced when changes were made to the software after its initial production and distribution.

## Context for Software Validation

Software verification and validation are difficult because a developer cannot test forever, and it is hard to know how much evidence is enough. It is necessary to develop a *level of confidence* that the device meets all requirements and user expectations. The necessary level of confidence depends on the safety risk imposed by the automated functions of the device.

## Differences between hardware and software

- ▶ The vast majority of software problems are traceable to errors made during the design and development process.
- ▶ Typically, testing alone cannot fully verify that software is complete and correct. In addition to testing, other verification techniques and a structured and documented development process should be combined to ensure a comprehensive validation approach.
- ▶ Unlike hardware, software is not a physical entity and does not wear out.
- ▶ Updates in software can introduce defects.
- ▶ Unlike some hardware failures, software failures occur without advanced warning.

- ▶ Seemingly insignificant changes in software code can create unexpected and very significant problems elsewhere in the software program.
- ▶ personnel who make maintenance changes to software may not have been involved in the original software development.
- ▶ Therefore, accurate documentation is essential.
- ▶ For these and other reasons, software engineering needs an even greater level of managerial scrutiny and control than does hardware engineering.

*Software testing is a necessary activity. However, in most cases software testing by itself is not sufficient to establish confidence that the software is fit for its intended use. In order to establish that confidence, software developers should use a mixture of methods and techniques to prevent software errors and to detect software errors that do occur.*

## Principles of software validation

- ▶ One needs a formal specification: The software validation process cannot be completed without an established software requirements specification
- ▶ Many new startup companies are under the false impression that we first develop the software in some fashion, and will worry about the FDA when it comes to validating it.

## Activities and tasks

- ▶ Quality planning
- ▶ Requirements
- ▶ Designs
- ▶ Construction, implementation or coding
- ▶ Testing by the developer
- ▶ User site testing
- ▶ Maintenance and software changes

## Managing external components

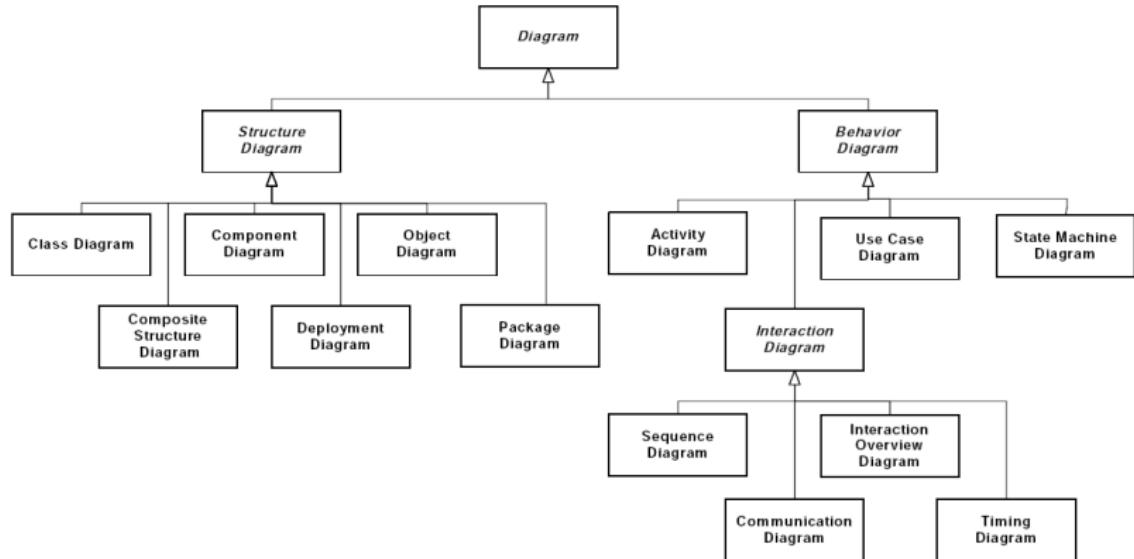
*Where the software is developed by someone other than the device manufacturer (e.g. libraries) the software developer may not be directly responsible for compliance with FDA regulations. In that case, the party with regulatory responsibility (e.g. the device manufacturer) needs to assess the adequacy of the off-the-shelf software developer's activities and determine what additional efforts are needed to establish that the software is validated for the device manufacturer's intended use.*

The *Unified Modeling Language* (UML) is a general-purpose, developmental, modeling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system:

*The Unified Modeling Language (TM) – UML – is OMG's most-used specification, and the way the world models not only application structure, behavior, and architecture, but also business process and data structure.*

Likewise building plans are created in many other areas, e.g. architects, machine design, electronic circuits, plans are also needed for the design and implementation of software systems.

<https://www.uml.org/>



The current UML standards call for 13 different types of diagrams:

**Structural UML diagrams:** Structure diagrams emphasize the things that must be present in the system being modeled:

**Class Diagram:** Class diagrams are the backbone of almost every object-oriented method, including UML. They describe the static structure of a system.

**Package Diagram:** Package diagrams are a subset of class diagrams, but developers sometimes treat them as a separate technique. Package diagrams organize elements of a system into related groups to minimize dependencies between packages.

**Component Diagram:** Component diagrams describe the organization of physical software components, including source code, run-time (binary) code, and executables.

**Deployment Diagram:** Deployment diagrams depict the physical resources in a system, including nodes, components, and connections.

**Behavioral UML diagrams:** Behavior diagrams emphasize what must happen in the system being modeled:

**Use-Case Diagram:** Use case diagrams model the functionality of a system using actors and use cases.

**State-Transition Diagram:** State-Transition diagrams, now known as state machine diagrams and state diagrams describe the dynamic behavior of a system in response to external stimuli. State diagrams are especially useful in modeling reactive objects whose states are triggered by specific events.

**Activity Diagram:** Activity diagrams illustrate the dynamic nature of a system by modeling the flow of control from activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. Typically,

activity diagrams are used to model workflow or business processes and internal operation.

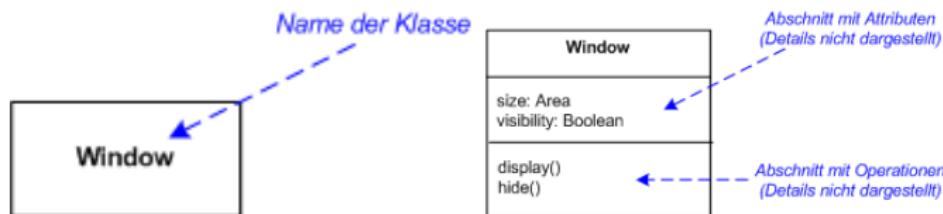
**Interaction diagram:** Interaction diagrams describe interactions among classes in terms of an exchange of messages over time.

**Timing Diagram:** A timing diagram is a type of behavioral or interaction UML diagram that focuses on processes that take place during a specific period of time. They're a special instance of a sequence diagram, except time is shown to increase from left to right instead of top down.

There are many tools available which could be used to draw UML diagrams, convert diagrams into code or vice versa. All tools have their pros and cons.

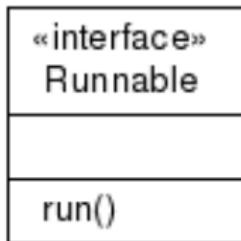
- ▶ Umbrello
- ▶ StarUML
- ▶ UMLet
- ▶ Visio
- ▶ Eclipse Papyrus (Eclipse Plugin)
- ▶ ArgoUML
- ▶ IntelliJ Diagrams?

The class diagram is the main building block of object-oriented modeling. It is used for general conceptual modeling of the structure of the application, and for detailed modeling translating the models into programming code. Class diagrams can also be used for data modeling. The classes in a class diagram represent both the main elements, interactions in the application, and the classes to be programmed.

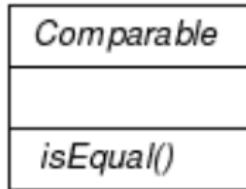


Source: Wikipedia

A class contains a **name**, **attributes** and **methods**. Attributes and methods could be hidden if they do not cover an important part in the system design.



Interfaces are similar to abstract classes. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface. Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types.



An abstract class is a class that is declared abstract. It may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

Window
size : Area
visibility : Boolean
display()
resize(scale : float)
getSize() : Area

An **attribute** is an individual thing that differentiate one object from another and determine the appearance, state, or other qualities of that object. Each object for a class has the given set of attributes with own values.

A **class attribute** is an attribute which exists only once for all objects. All objects of a given class share the same attribute. Class attributes are underlined.

Window
size : Area
visibility : Boolean
display()
resize(scale : float)
getSize() : Area

A **method** is a set of statements to perform an operation.

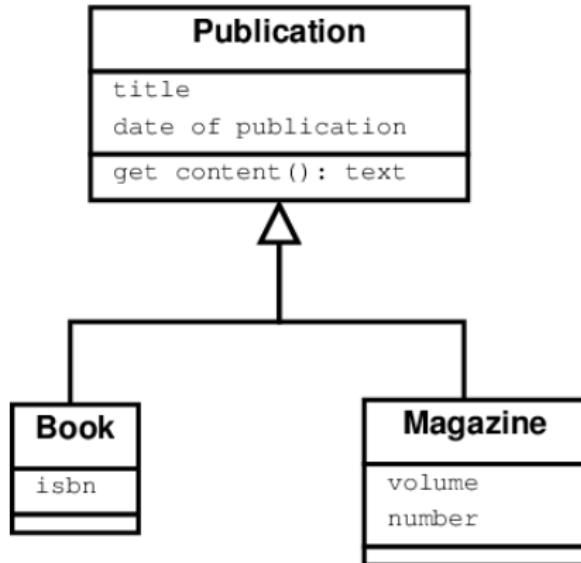
Sometimes, a method is also called member function.

A **class method** is an operation which does not depend on any object of that class. This method could also be called if there is no object of the class (e.g. utility method), therefore a class method could not access any attributes (expect class attributes).

Toolbar
# currentSelection : Tool
# toolCount : Integer
+ pickItem(i : Integer)
+ addTool(t : Tool)
+ removeTool(i : Integer)
+ getTool() : Tool
# checkOrphans()
- compact()

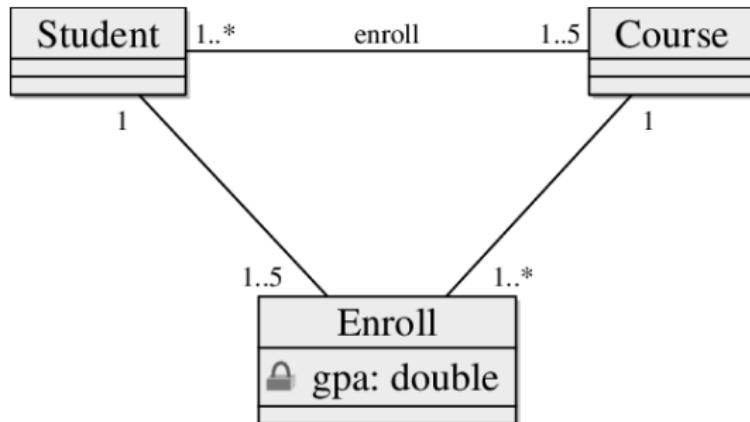
There are 3 different types of visibility for attributes and methods:

- Private only visible within the class
- # Protected visible in the class itself and all inherited classes
- + Public visible from everywhere, no protection.



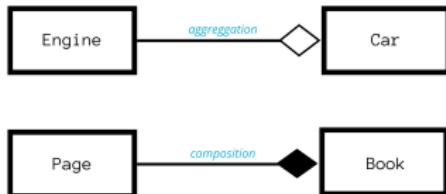
Source: [sourcemaking.com](http://sourcemaking.com)

In object-oriented programming, inheritance is the mechanism of basing one class upon another class, retaining similar implementation. Also defined as deriving new classes (sub classes) from existing ones (super class) and forming them into a hierarchy of classes. An object created through inheritance acquires all the properties and behaviours of the parent object (except: constructors, and some other language specific constructs). Inheritance allows programmers to create classes that are built upon existing classes, to specify a new implementation while maintaining the same behaviours (realizing an interface), to reuse code and to independently extend original software via public classes and interfaces.



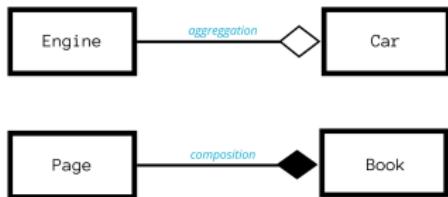
Source: [www.researchgate.net](http://www.researchgate.net)

An **association** defines a relationship between classes of objects that allows one object instance to cause another to perform an action on its behalf. This relationship is structural, because it specifies that objects of one kind are connected to objects of another and does not represent behaviour.



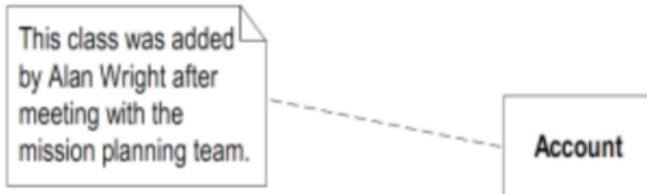
With an aggregation, the child can exist independently of the parent.

So thinking of a Car and an Engine, the Engine doesn't need to be destroyed when the Car is destroyed.



Composition implies that the contained class cannot exist independently of the container. If the container is destroyed, the child is also destroyed.

Take for example a Page and a Book. The Page cannot exist without the Book, because the book is composed of Pages. If the Book is destroyed, the Page is also destroyed.



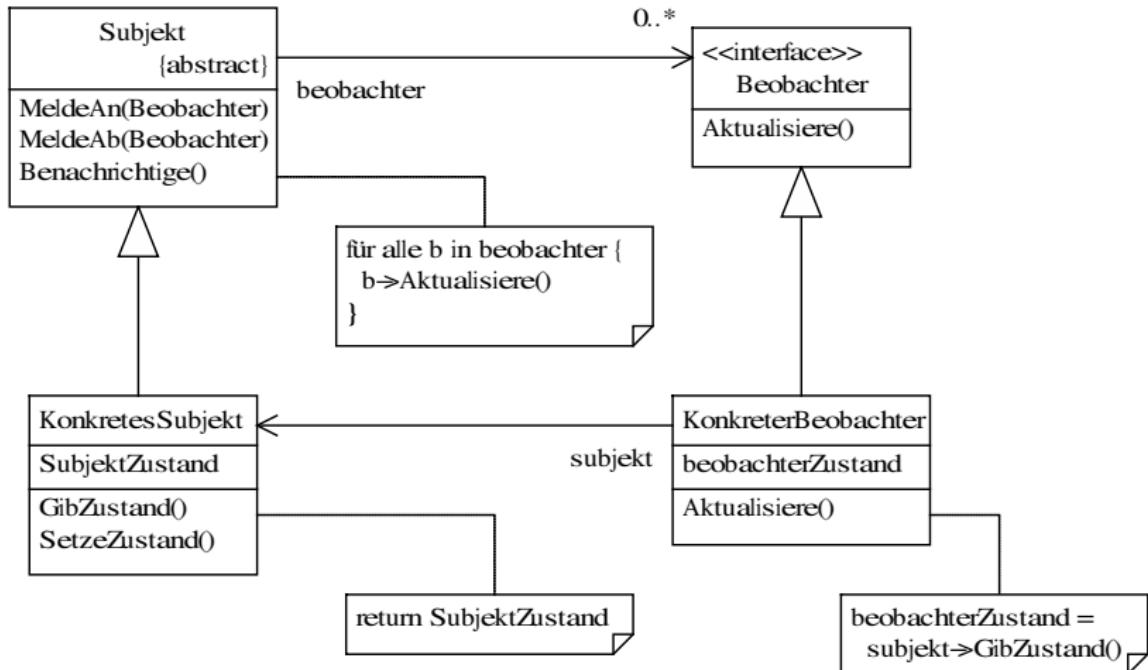
Source: [www.stackoverflow.com](http://www.stackoverflow.com)

A comment is a textual annotation that can be attached to a set of elements.

A comment gives the ability to attach various remarks to elements. A comment carries no semantic force, but may contain information that is useful to a modeler. A comment may be owned by any element.

# Class Diagram Example: Observer

393/418



1. identify classes and their relations,
2. add relevant attributes and operations,
3. generalize using inheritance,
4. group classes into modules

Ausgangspunkt ist den meisten Fällen eine verbale Beschreibung des Problembereichs im Vokabular des Anwenders. Eine gängige, wenn auch zum Teil kritisierte, Methode zur Identifizierung der Klassen (und Objekte) basiert auf einer Textanalyse, bei der die Substantive als mögliche Kandidaten dienen:

Personen, Rollen

Lieferant, Kunde, Mitarbeiter, Student, Dozent

Dinge, Geräte

Behälter, Tank, Regler, Pumpe, Zeitgeber, Schalter, Messgerät, Sensor, Filter, Anzeige

Abstraktionen

Stack, Array, Tabelle, Sequenz, Menge, Figur, Dreieck, Linie, Datei, Baustein, Element

Beschreibungen, Dokumente  
Ereignisse, Vorgänge

Rezeptur, Produkt, Konto  
Prozess, Simulation, Steuerung, Nachricht

Interaktionen

Bestellung, Kaufvertrag, Buchung, Reservation, Anmeldung, Au

In der Regel findet man in einer ersten Analyse des Problembereichs mehr Klassen, als man wirklich benötigt. Auf weiter zu verwendende Klassen müssen die folgenden Aussagen zutreffen:

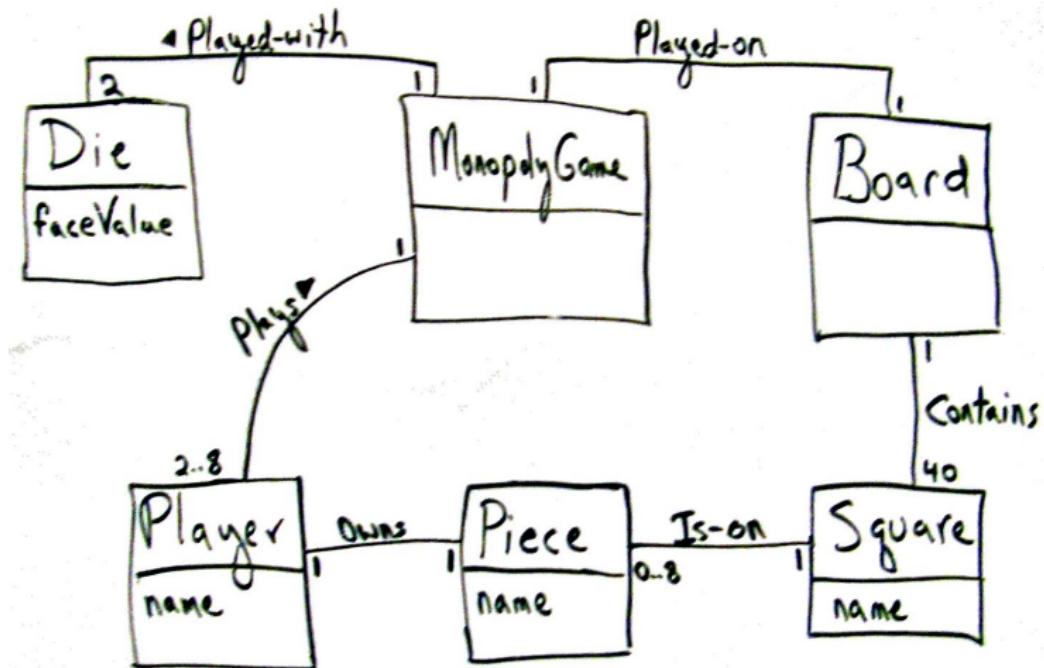
1. Ihr Name ist eindeutig, d.h. kein Synonym einer bereits ausgewählten Klasse.
2. Sie enthält Informationen (Attribute), die innerhalb des Systems gespeichert werden sollen.
3. Sie stellt system-relevante Operationen zur Verfügung (mehr als nur Setzen und Abfragen).

Beziehungen zwischen den Objekten entsprechen physischen oder logischen Verknüpfungen mit zugehörigen Kardinalitäten analog den Relationen zwischen Entitäten:

- ▶ Assoziation: semantische Verknüpfung
- ▶ Aggregation: “hat-ein”-Verknüpfung
- ▶ Verwendung: Nutzungs-Verknüpfung
- ▶ Vererbung: “ist-ein”-Verknüpfung

# Example: Class Diagram of a Game

398/418



Source: Craig Larman

- ▶ Attribute sind die Eigenschaften der Objekte:
  1. Wie wird das Objekt allgemein beschrieben?
  2. Wie wird das Objekt im Problembereich beschrieben?
  3. Können zusammengehörende Attribute zu einer Gruppe zusammengefasst werden? (Bsp: Vorname, Nachname, Anrede ergibt Name)
- ▶ Methoden sind die Dienste, die ein Objekt bereitstellt:
  1. Berechnungen
  2. Überwachung
  3. Kommunikation

aus Gründen der Übersichtlichkeit sollen nur explizite und keine impliziten Methoden im Objektmodell dargestellt werden. Implizite Methoden sind: Konstruktorfunktionen, Zugriffsfunktionen zu den Attributen, Verbindungsfunctionen zum Setzen und Lösen von Verbindungen.

Durch eine Vererbungsstruktur werden Klassen die

- ▶ Spezialfälle einer andern Klasse sind oder
- ▶ gemeinsame Attribute mit andern Klassen haben

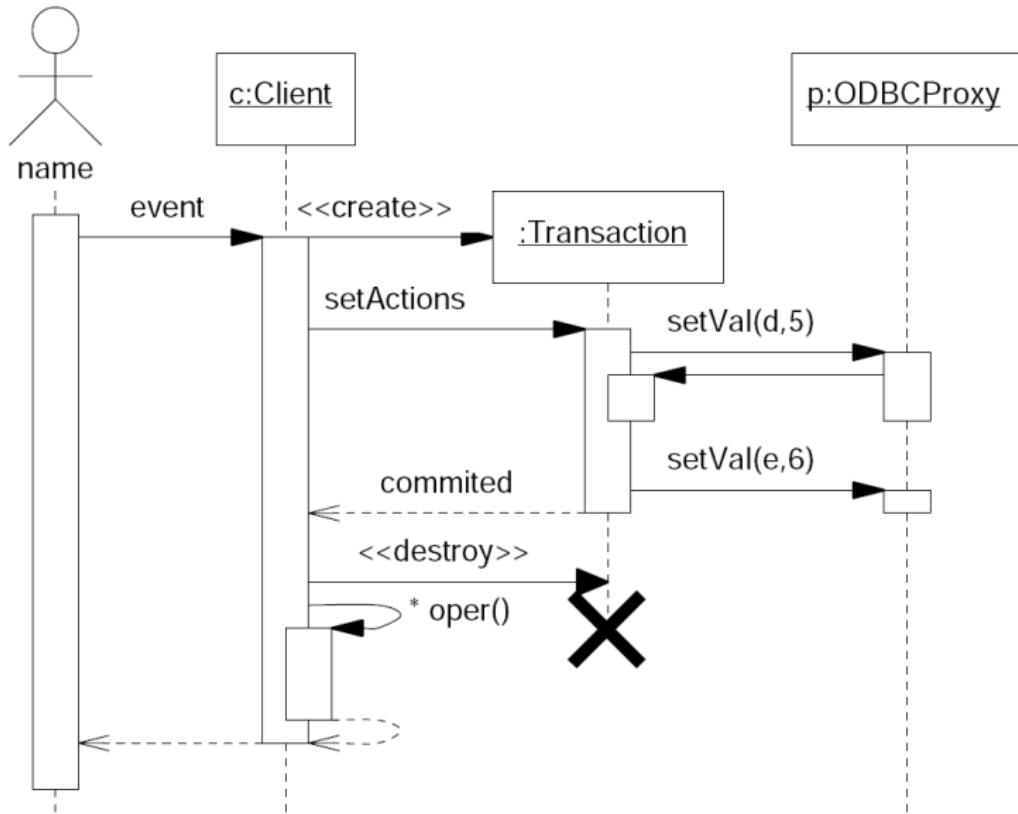
in einer gemeinsamen Basisklasse zusammengefasst.

- ▶ Jede Klasse ist Teil genau eines Moduls (Package).
- ▶ Die Kopplung zwischen Klassen unterschiedlicher Kategorien ist minimal.

Interaction diagrams are models that describe how a group of objects collaborate in some behavior - typically a single use-case. The diagrams show a number of example objects and the messages that are passed between these objects within the use-case.

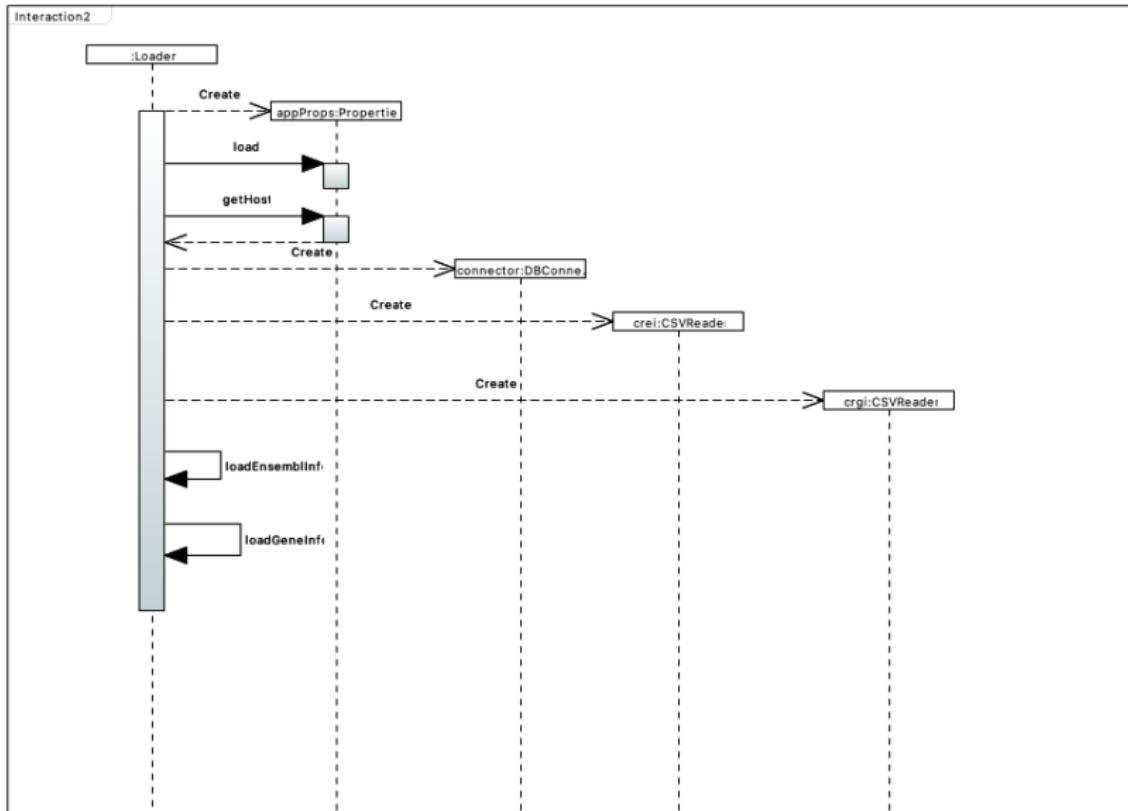
# Sequence Diagram

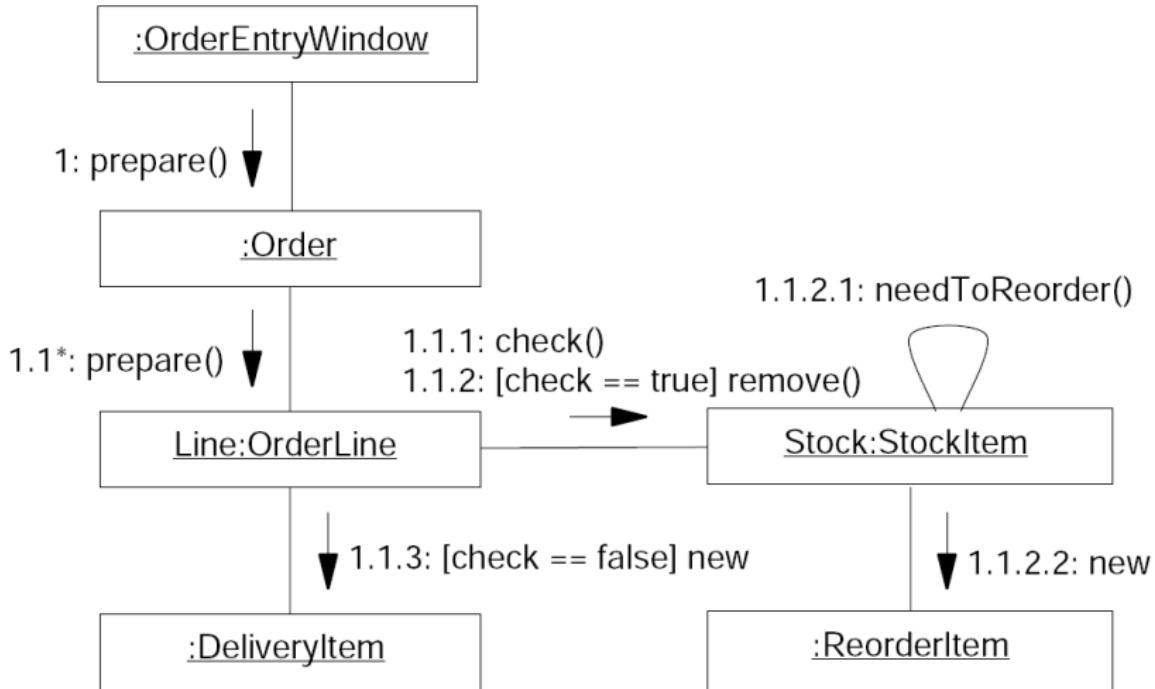
403/418



# Example: Gene Loader

404/418

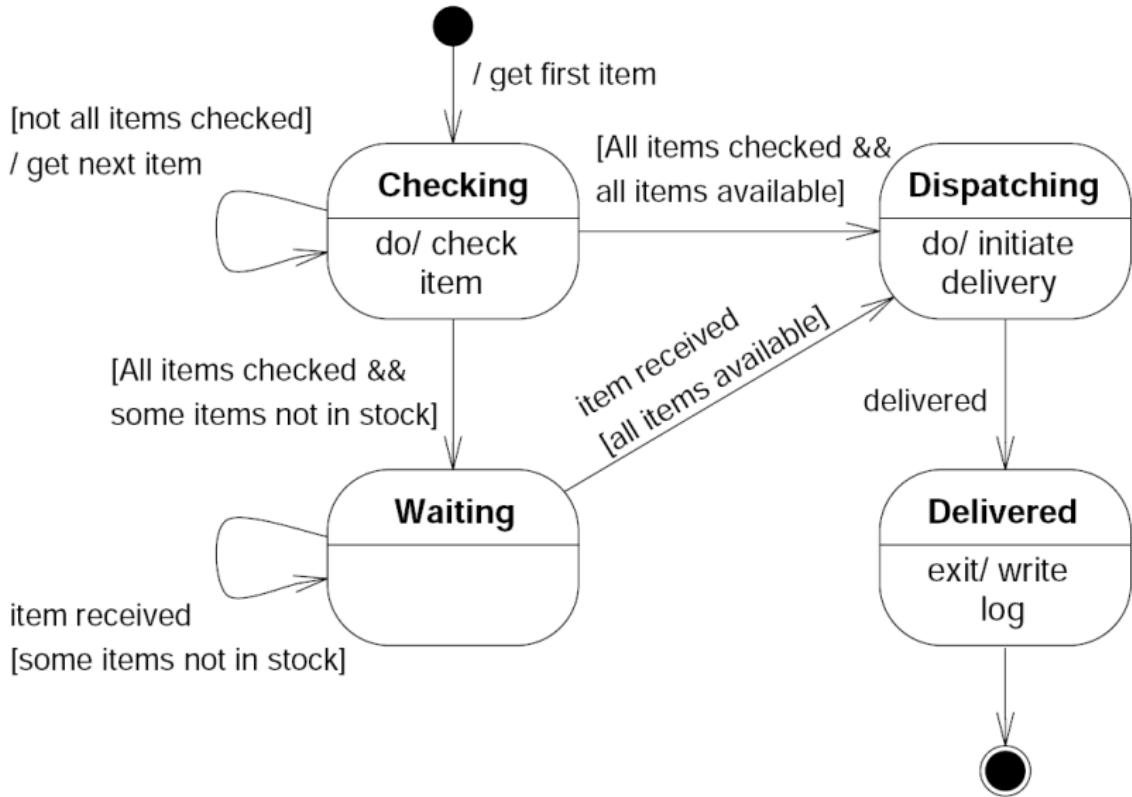




1. Szenarien entwerfen
2. Ereignisse und betroffene Objekte identifizieren
3. Ereignispfad skizzieren

# State Machine Diagram

407/418





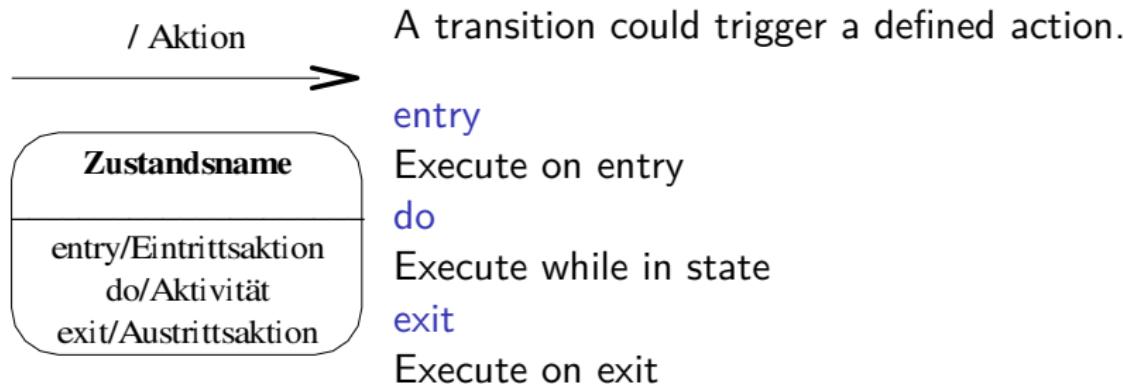
The initial state of a state machine diagram, known as an initial pseudo-state, is indicated with a solid circle. A transition from this state will show the first real state.



The final state of a state machine diagram is shown as concentric circles. An open loop state machine represents an object that may terminate before the system terminates, while a closed loop state machine diagram does not have a final state; if it is the case, then the object lives until the entire system terminates.

State1

A state is denoted by a round-cornered rectangle with the name of the state written inside it.



Ereignis



A transition is a change in one state into another state which is occurred because of some event. A transition causes a change in the state of an object.

Ereignis[Wächter]



A transition could be extended with a guard.  
This is a boolean condition. The transition  
will only be fulfilled, if the condition is true.

An event could be

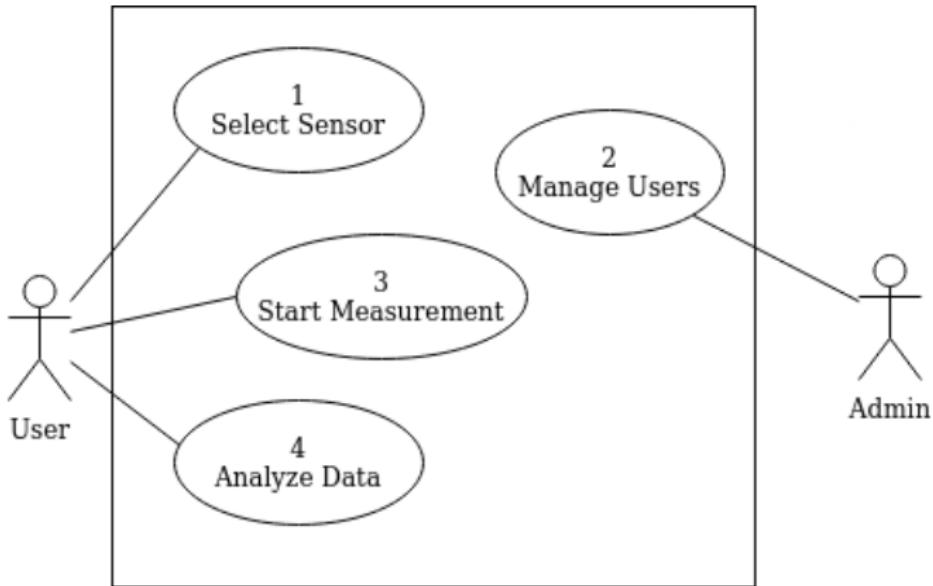
- ▶ a condition evaluates to true
- ▶ calling a method
- ▶ elapsed timeframe
- ▶ reaching a timepoint

If an event happens and the object is not able to react on that event, then the event will be ignored.

1. Identify the initial state and the final terminating states.
2. Identify the possible states in which the object can exist (boundary values corresponding to different attributes guide us in identifying different states).
3. Label the events which trigger these transitions.

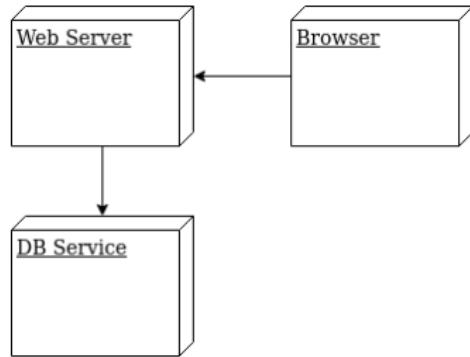
# Use Case Diagram

415/418



1. Identify who is going to be using the system directly. These are the Actors.
2. Pick one of those Actors.
3. Define what that Actor wants to do with the system. Each of these things that the actor wants to do with the system become a Use Case.
4. For each of those Use Cases decide on the most usual course when that Actor is using the system. What normally happens.
5. Describe that basic course in the description for the use case.
6. Once you're happy with the basic course now consider the alternatives and add those as extending use cases.

Use Case Name		[Name of the use case]
Actors		[An actor is a person or other entity external to the system being specified who interacts with the system and performs use cases to accomplish tasks]
Preconditions		[Activities that must take place, or any conditions that must be true, before the use case can be started]
Normal Flow	Description	[User actions and system responses that will take place during execution of the use case under normal, expected conditions.]
	Postconditions	[State of the system at the conclusion of the use case execution with a normal flow (nominal)]
Alternative flows and exceptions		[Major alternative flows or exceptions that may occur in the flow of event]
Non functional requirements		[All non-functional requirement: e.g., dependability (safety, reliability, etc.), performance, ergonomic]



The deployment diagram maps the software architecture to the physical system architecture that executes it. In distributed systems, it models the distribution of the software across the physical nodes.