



















**Software Engineering
in der industriellen Praxis
(SEIP)**

Dr. Ralf S. Engelschall

Research RE Crawling the problem domain's body of knowledge to find starting points. 	Abstraction AB Solving the problem in a model of the problem before applying it to the real problem to get a better understanding. 	Lateral Thinking LT Approaching the problem indirectly and creatively to find a not obvious solving lever. 	Backward Search BS Looking at the expected results and determine which operations could bring you to them. 
Brainstorming BR Suggesting larger number of solution ideas for further combination and development. 	Generalization GE Thinking about the problem more abstract to get rid of special cases. 	Hypothesis Proof HP Assuming a possible solution and trying to prove (or disprove) the assumption to find starting points. Q.E.D. 	Backtracking BT Remembering path towards the solution and on failure tracking back and choosing a new path. 
Analogy AN Thinking in terms of similar problems for which solutions are known to get inspired. 	Specialization SP Solving a special case first to get an impression towards the full solution. 	Root Cause RC Asking "Why?" five times in sequence to explore the cause-and-effect relationships underlying the problem. 5x WHY? 	Divide & Conquer DC Breaking down the large complex problem into smaller, easier solvable partial problems. 
Reduction RD Transform the problem into another one for which a solutions already exists to reduce solving efforts. 	Variation VA Changing the problem context or expressing the problem differently to find a not obvious solving lever. 	Means End ME Choosing an action from scratch just at each step to move closer and closer to the solution. 	Trial & Error TE As a last resort, brute-force testing all potential solutions in case of a small enough total solution space. 

Definition: **Heuristic** — fallible experience-based technique or strategy for problem solving in case *Rule of Thumb*, *Guessing*, *Intuitive Judgement*, *Common Sense* and *Stereotyping* are either not sufficient or not appropriate.

The **Problem Solving Heuristics** are experience-based techniques or strategies that can be used for problem-solving when other approaches do not make any progress.

The heuristics are mainly used for inspiration so that you don't spend too long and instead find a new starting point for solving the problem ("If you find yourself in a hole, stop digging!").

With **Research** one searches for facts, with **Brainstorming** one proposes a large number of spontaneous solution ideas, in **Analogy** one thinks of similar problems that have already been solved, and in **Reduction** one transforms the problem into an already solved problem.

With **Abstraction** one solves the problem first in a more abstract model, with **Generalization** one generalizes the problem to one with fewer special cases, in **Specialization** one tries to be inspired by a special case, and in **Variation** one tries to change one's perspective on the problem.

In **Lateral Thinking** one approaches the problem in a deliberately indirect and creative way, with **Hypothesis Proof** one searches for a solution by proof for a possible or not possible fictitious solution, with **Root Cause** one goes to the root of the problem step by step and in **Means End** one tries to approach the solution in small steps.

In **Backward Search** one tries to get from a fictitious solution backward on the way to the solution, with **Backtracking** one chooses a partly new path in the direction of the solution in case of a failure, in **Divide & Conquer** one breaks down the big problem into smaller and easier to solve subproblems, and with **Trial & Error** one tries all solution combinations as a last resort and/or if the solution space is small enough.

Questions

- When is the rather mundane **Problem Solving Heuristic** called **Trial & Error** acceptable?

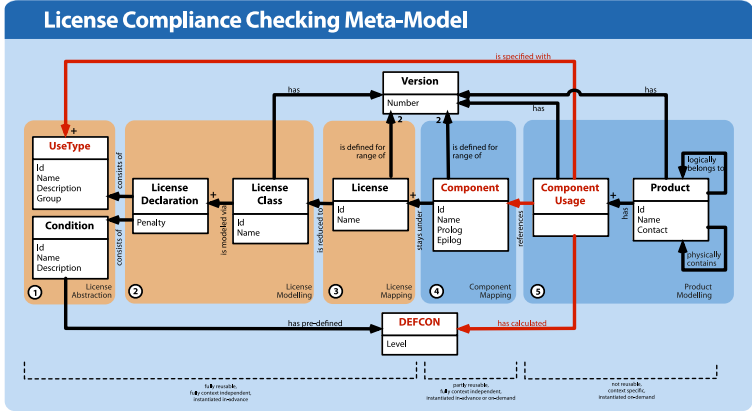
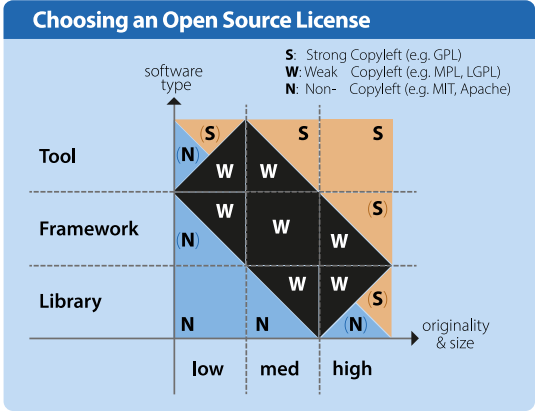
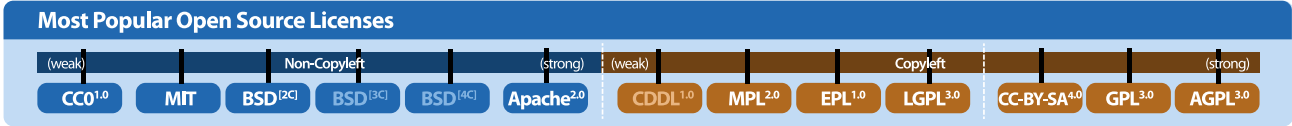
Open Source Definition

Distribution terms (license) of Open Source Software must be compliant with the following criterias:

- Free Redistribution
- (Original) Source Code (Availability)
- Derived Works (Allowance)
- Integrity of the Author's Source Code
- No Discrimination Against Persons or Groups
- No Discrimination Against Fields of Endeavor
- Distribution of (Non-Exclusive) License
- License Must **Not Be Specific to a Product**
- License Must **Not Restrict Other Software**
- License Must Be **Technology-Neutral**

Open Source Personality Streams

§ Software Sharing	Dogmatism Social Equity Politics	Industry Private Science
@ Software Hacking	Fundamentalism Art Hacking	
€ Software Engineering	Pragmatism Business Engineering	



Open Source Software is software that has been placed under an **Open Source License**. All licenses recognized as **Open Source Licenses** meet the **Open Source Definition**, which states, among other things, that the software may be freely distributed in source code and changed and that there is no discrimination of persons, groups, or purposes.

In practice, one knows three **Open Source Personality Streams**: **Software Sharing** with dogmatic and political persons, who are fighting for social justice; **Software Hacking** with fundamental and artistic persons, who develop software with maximum ambition; and **Software Engineering** with pragmatic people who use the software in practice.

There are hundreds of **Open Source Licenses**. However, one can split them into a few classes and sort them according to their strength, i.e., how strongly they protect the software itself. One distinguishes generally between licenses without and with a so-called **Copyleft** effect. This consists of license clauses in order to keep the original software free (in the sense of freedom and availability, not free of charge) and additionally to keep all modifications and extensions to the software also free.

The weakest license in practice is **Creative Commons Zero (CC0)** (or **Public Domain**), which effectively allows anyone to do anything.

The strongest license is the **Affero General Public License (AGPL)**, which protects software even in the case of use in the form of Software as a Service (SaaS). At the Copyleft boundary is the **Apache License**, which does not yet have a Copyleft effect but still tries to maximally protect the software and the originator.

In practice, a distinction is made between licenses with no, weak and strong Copyleft. To decide for a software under which class of license one publishes it, one differentiates between two dimensions: on the one hand, the type of software (**Tool**, **Framework** or **Library**) and on the other hand, the level of creation of the software. A **Tool** or a **Framework** with a medium or high level of creation is usually under weak or even strong Copyleft to protect the software and the author to the maximum. A **Library** or a **Framework** with a medium or low level of intellectual property is under a weak or even no Copyleft in order to achieve a maximum distribution of the software.

Questions

What do you call the effect in licenses of **Open Source Software**, in which the software remains free (in the sense of freedom and availability, not in the sense of free of charge) and additionally all modifications and extensions remain free as well?

Specification (Example)

Customer: Twitter Inc.
Business: Microblogging

Use-Cases 1/3 (profile):
- user can register an account
- user can "follow" other users
- user can create lists of users he follows

Use Cases 2/3 (send):
- user can send tweets
- tweets are based on words, each either a text "example", tag "#example", user reference "@example" or URL http://example.com
- tweets are either public broadcast or personal/direct messages
- user can re-tweet a message of others

Use Cases 3/3 (query):
- user can view timeline (chronological tweets of others he follows)
- user can search for tweets (by keyword "foo", tag "#foo", or user "@foo")
- user can view tag cloud

Frontends/Clients:
- mobile app (iOS, Android)
- desktop app (Windows, Mac OS X)
- web app
- embedded web widget (query use cases only)

Current Demand (as of 2012):
- 140M user profiles
- 400M tweets/day
- 6393 tweets/second peak
- 140 characters/tweet
- 30K queries/second
- 300 GB/hour data in total
- 4,4 tweets/day/user on average
- 103,4 follower/user
- < 5s tweet-write-to-read-delay

Future Demand:
- quadratic user and traffic growth

Conversion & Normalization

Calculation (Example)

Twitter Information		Traffic Bandwidth	
6.393 tweets/second peak	140 chars/tweet	350% overhead HTTP+TCP+IP+Ethernet	10000 Mbps
400.000.000 tweets/day (write)	4.630 tweets/second (write)	2,2 MB/s (write)	1250 MB/s
2.592.000.000 queries/day (read)	30.000 queries/second (read)	140,2 MB/s (read)	1000 Mbps
6,5 factor read/writes	10 tweets/query	142,4 MB/s	100 Mbps
	4,4 tweets/day/user		10 Mbps
	2,4 tweets/day (M. Fowler)		
	0,8 tweets/day (R. Engelschall)		
621.880.000 tweets/day (average) total	621.880.000 tweets/day (average) total		
64,3% users are active at all	64,3% users are active at all		
277.778 users/minute active	277.778 users/minute active		

Storage Requirements (static)	Storage Requirements (dynamic)	Computing Hardware Requirements
140.000.000 user profiles	300 GB/hour data in total	2000 requests/sec (read) AS performance
52.000 chars/users for profile	214 TB/month data in total	100 requests/sec (write) AS performance
6,62 TB profile (total)		150 servers for writes
103,4 follower/user	200% overhead storage	46,3 servers for reads
14.474.600.000 user follow links	1265,9 KB/s tweets	
32 bytes/link	104,3 GB/day tweets	
0,42 TB links (total)	3,1 TB/month tweets	
	200 chars/log entry	
	6763,6 KB/s log	
	557,3 GB/day log	
	16,6 TB/month log	
	9,2% ratio business data	
	90,8% ratio infrastructure data	

To get a better "feeling" for the scope and the degree of difficulty of an architecture to be developed, it is a good idea to do a **Back of the Envelope Calculation** ("rough calculation"). The method is as follows: in a spreadsheet, a two-column table is created in which the first column contains the number and the second column contains the unit.

Now, in the first step, the **Business-Given Key Figures**, i.e., the technically known numbers, are entered into the table as the first rows. They get the first color for differentiation.

Because these numbers usually do not tell enough, in a second step, different **Technology-Given Key Figures** are entered as rows. These may be taken from existing catalogs or are available from your own experience. They are given the second color for differentiation and serve above all as comparative figures to the **Business-Given Key Figures**.

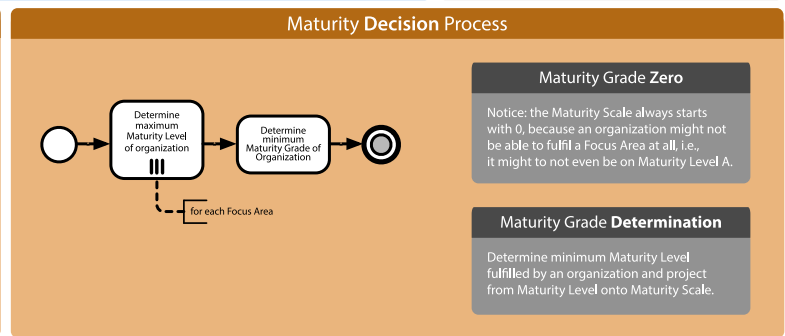
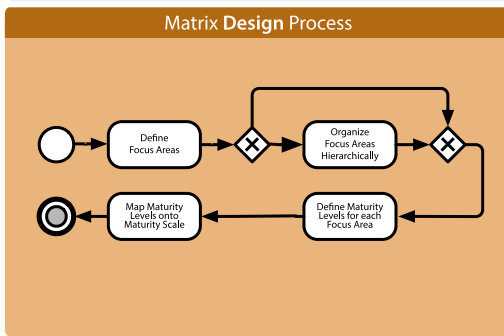
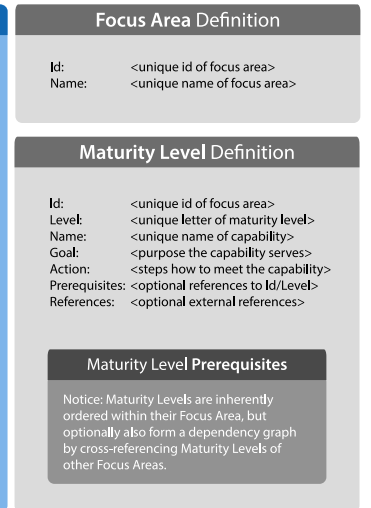
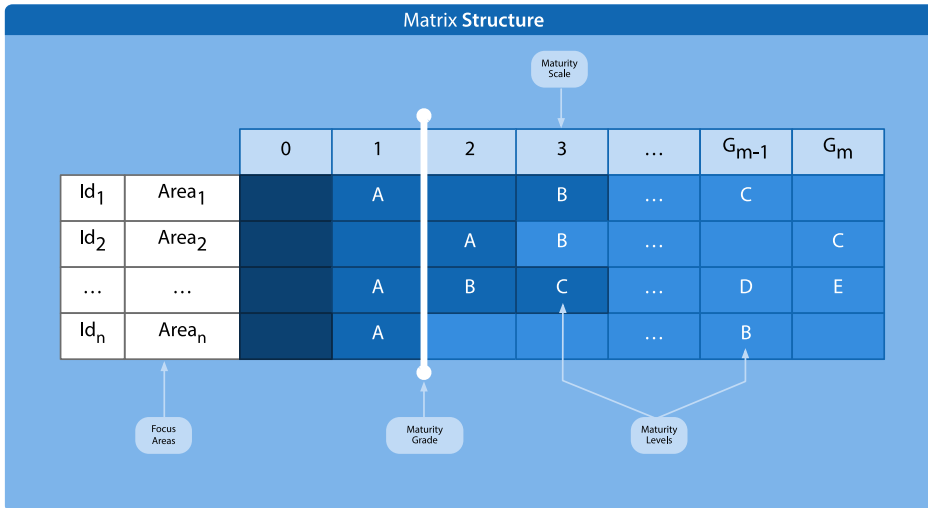
In the third step, both the **Business-Given Key Figures** and the **Technology-Given Key Figures** are compared to each other. The intermediate results, called **Intermediate Calculated Figures**, are spreadsheet cells with formulas, which get the third color to distinguish them.

Whenever an **Intermediate Calculated Figure** (or possibly already an **Business-Given Key Figure** or a **Technology-Given Key Figure**) provides a decisive hint or insight, you change the row to the fourth color. If this insight has potential relevance for the subsequent architecture (and thus represents a key point), the row is changed to the fifth color, which shows the **Resulting Architecture Crux Figure**.

Afterwards, you can optionally bundle the different rows into logical groups in the spreadsheet to make the spreadsheet clearer.

Questions

- What method can be used to get a better "feel" for the scope and difficulty of an architecture to be developed?



The Focus Area Maturity Model (FAMM) is a method to assess the maturity of an organization with respect to a specific topic area.

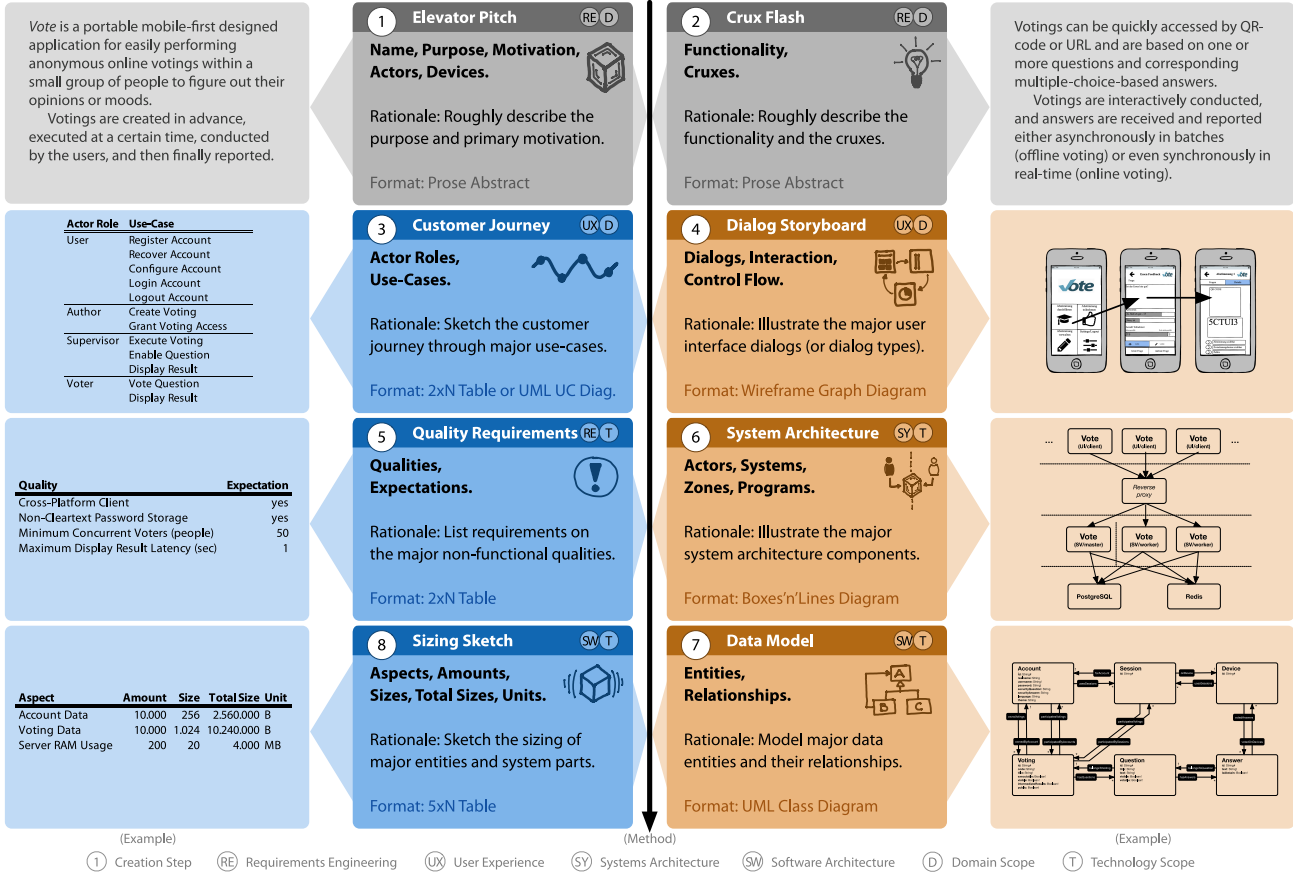
The structure of a FAMM is a matrix of horizontal Focus Areas and their their Maturity Levels and possible vertical Maturity Grades on a Maturity Scale. Per Focus Area there can be one or any number of Maturity Levels and their positions on the Maturity Scale are based on the importance of the Focus Areas and the relationship between the Focus Areas and their Maturity Levels. This matrix is designed in a first step for a topic area and is then fixed.

In order to determine the Maturity Level of an organization, determine for each Focus Area the maximum Maturity Level the organization fulfills. The Maturity Level of the organization is then derived from the minimum Maturity Level across all Focus Areas and the projection of this Maturity Level onto the Maturity Scale.

Since the Maturity Levels should only ever be positioned in the matrix above Maturity Grade 0, in the worst case, an organization has a Maturity Grade 0 if it does not fulfill a Focus Area at all.

Questions

- ? From whom can you determine the maturity level with the Focus Area Maturity Model (FAMM)?



In order to understand a Business Information System to be created at a very early stage, a so-called **Big Picture** should be created. This can be developed according to the method **8-D** of Dr. Ralf S. Engelschall, which consists of 8 dimensions (4 domain-specific and 4 technical ones).

In the first step, an **Elevator Pitch** is created, which is defined by **Name, Purpose, Actors** and **Devices** to give a rough overview of the solution. In the second step, an additional **Crux Flash** is created, which roughly describes the functionality and its main cruxes. Both steps usually consist of only a few prose sentences.

In the third step, a **Customer Journey** is outlined, which is describe, via **Actor Roles** and **Use Cases**, which use cases the different actor roles experience. This usually consists of only a 2-column table.

In the fourth step, a **Dialog Storyboard** is illustrated, which is shown via **Dialogs, Interactions** and **Control Flow**, and which illustrates the user interface the application offers. This usually consists of a graph of **Wireframes** (intentionally very rough and fuzzy sketches of the dialogs).

In the fifth step, the **Quality Requirements** are listed via **Qualities** and **Expectations**, to show which expectations exist concerning non-functional quality properties. These are usually executed as a 2-column table.

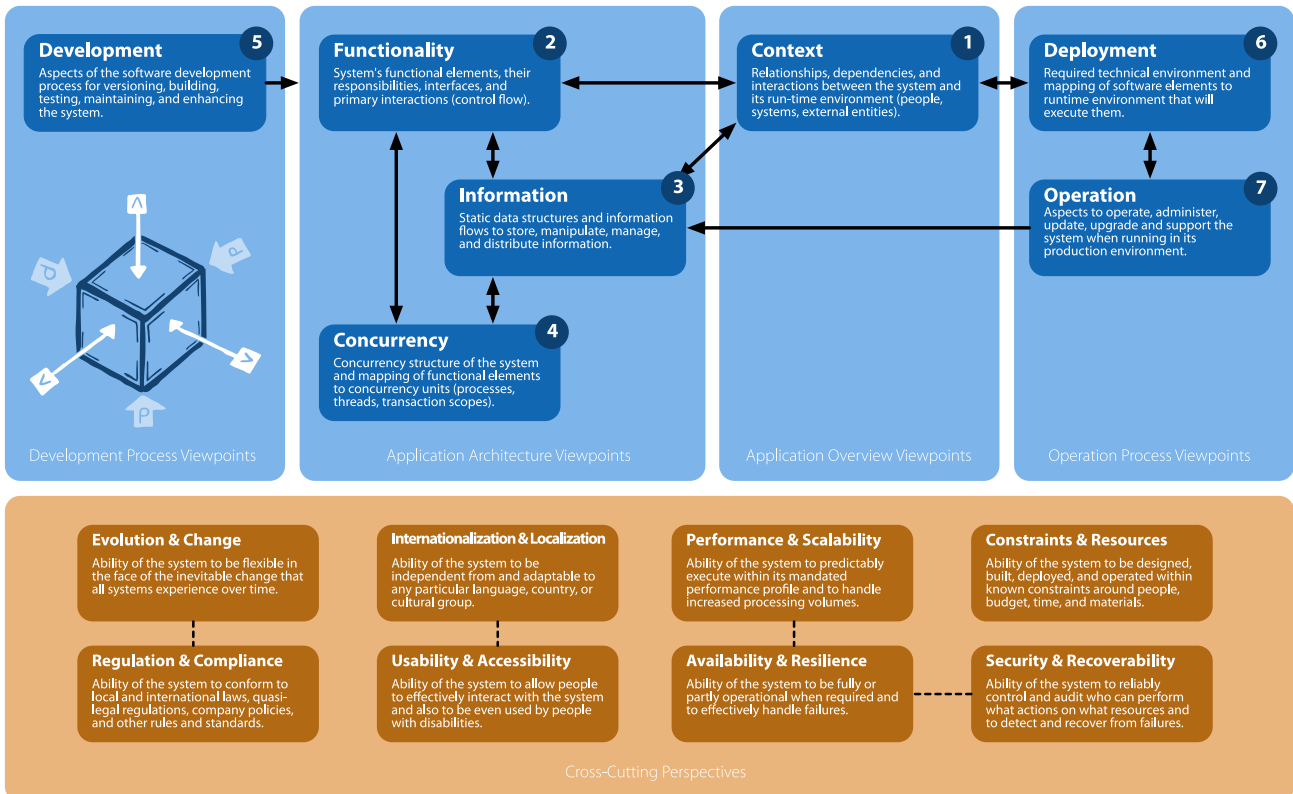
In the sixth step, the **System Architecture** is illustrated, which via **Actors, Systems, Tiers/Areas** and **Programs** shows, which main components will exist on the level of System Architecture. This usually consists of a "Boxes'n'Lines" diagram.

In the seventh step, the **Data Model** is modeled, which, via **Entities** and **Relationships**, shows which main domain-specific classes and relationships the data of the application has. This is usually represented as a "UML Class Diagram".

In the eighth and last step, a **Sizing Sketch** is created, which shows via **Aspects, Amounts, Sizes, Total Sizes** and **Units**, which orders of magnitude (in the "worst case") are to be expected with the data and the system components. This usually consists of a 5-column table.

Questions

- ❓ What are the eight dimensions of the **8-D** model supposed to do?



In order to create an **Architecture Description** (in the German context usually called **IT-Konzept**) for an application, one documents methodically via **Viewpoints** and **Perspectives**.

The former are 2-dimensional diagrams with an explanation of a single specific fact. The latter are explanations of an overall issue, all of which are derived from **Non-Functional Requirements**.

How many **Viewpoints** and **Perspectives** actually need to be documented depends on the **Concern** of the **Stakeholders**! A standard set of **Viewpoints** and **Perspectives** is the following.

For the **Viewpoints** it makes sense to document for instance 7 particular **Views**: **Context**, **Functionality**, **Information**, **Concurrency**, **Development**, **Deployment** and **Operation**. The two viewpoints **Functionality** and **Information** are the two most important ones to document the architecture of an application. They should always be documented and, therefore, never omitted!

For the **Perspectives**, it makes sense to document for instance 8x2 particular **Aspects** which are based on common **Non-Functional Requirements**: **Evolution & Change**, **Regulation & Compliance**, **Internationalization & Localization**, **Usability & Accessibility**, **Performance & Scalability**, **Availability & Resilience**, **Constraints & Resources** and **Security & Recoverability**. For the **Perspectives** in practice one has to take into account all the relevant **Non-Functional Requirements** relevant to the project and must not limit oneself to just these usual ones.

Questions

- ? Which two **Viewpoints** in the **Architecture Description** of an application should always be documented?